A DISTRIBUTED FRAMEWORK FOR SITUATION AWARENESS ON CAMERA NETWORKS

A Thesis
Presented to
The Academic Faculty

by

Kirak Hong

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science
College of Computing

Georgia Institute of Technology
August 2014

A DISTRIBUTED FRAMEWORK FOR SITUATION AWARENESS ON CAMERA NETWORKS

Approved by:

Professor Umakishore Ramachandran,
Committee Chair
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor Umakishore Ramachandran,
Advisor
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor Santosh Pande
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor Mostafa H. Ammar
School of Computer Science
College of Computing
Georgia Institute of Technology

Professor Bharat Jayaraman
Department of Computer Science and Engineering
University at Buffalo, The State University of New York

Professor Liviu Iftode
Department of Computer Science
Rutgers University

Date Approved: 27 June 2014

To

my amazing wife, Kihyun Kim,

who I am lucky to be with,

my parents, Sungsam Hong and Wonsuk Ha,

who encouraged my journey with warm smiles,

my sister, Sunju Hong,

who has been my emotional anchor.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to my advisor, Professor Umakishore Ramachandran, whose insightful advice and strong support helped me to achieve this work. Although I arrived at Georgia Tech with little experience in academic research, he guided me with patience and encouraged me to dive into research problems with no fear. I can never overstate the importance of his support during my PhD study.

My committee members — Professors Mostafa Ammar, Santosh Pande, Bharat Jayaraman, and Liviu Iftode — provided valuable and constructive feedback on my thesis work. In particular, I appreciate Professor Bharat Jayaraman for his thoughtful advice on the Spatio-temporal Analaysis project. I would also like to offer a special appreciation to Professor Liviu Iftode for his keen comments on the Target Container project.

Dr. Boris Koldehofe and Beate Ottenwälder from University of Stuttgart helped me in the Opportunistic Event Processing and Mobile Fog projects. With their solid knowledge of complex event processing, I could shape and tackle the research problems in more interesting ways.

I would like to extend my appreciation to my labmates in my advisor's research group, Dr. Hyojun Kim, Dr. Lateef Yusuf, Dave Lillethun, Dushmanta Mohapatra, Moonkyung Ryu, Wonhee Cho, and Yeonju Jeong. With their support and friendship, my life at Georgia Tech was always exciting and pleasing.

Personally, I want to thank my parents and wife who made me to go through this stage of my life. My parents motivated me to find and solve problems by myself from an early age, while always being my constant. My deepest appreciation to my wife, who has been the strongest support for my study. Her cheerful voice raised me up when I was down, and her lovely smile lighted my way when I was lost.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

With the proliferation of cameras and advanced video analytics, situation awareness applications that automatically generate actionable knowledge from live camera streams has become an important class of applications in various domains including surveillance, marketing, sports, health care, and traffic monitoring. However, despite the wide range of use cases, developing those applications on large-scale camera networks is extremely challenging because it involves both compute- and data-intensive workloads, has latency-sensitive quality of service requirement, and deals with inherent dynamism (e.g., number of faces detected in a certain area) from the real world. To support developing large-scale situation awareness applications, this dissertation presents a distributed framework that makes two key contributions: 1) it provides a programming model that ensures scalability of applications and 2) it supports low-latency computation and dynamic workload handling through opportunistic event processing and workload distribution over different locations and network hierarchy.

To provide a scalable programming model, two programming abstractions for different levels of application logic are proposed: the first abstraction at the level of real-time target detection and tracking, and the second abstraction for answering spatio-temporal queries at a higher level. The first programming abstraction, Target Container (TC), elevates target as a first-class citizen, allowing domain experts to simply provide handlers for detection, tracking, and comparison of targets. With those handlers, TC runtime system performs priority-aware scheduling to ensure real-time tracking of important targets when resources are not enough to track all targets. The second abstraction, Spatio-temporal Analysis (STA) supports applications to

answer queries related to space, time, and occupants using a global state transition table and probabilistic events. To ensure scalability, STA supports bounded communication overhead of state update by providing tuning parameters for the information propagation among distributed workers.

The second part of this work explores two optimization strategies that reduce latency for stream processing and handle dynamic workload. The first strategy, an opportunistic event processing mechanism, performs event processing on predicted locations to provide just-in-time situational information to mobile users. Since location prediction algorithms are inherently inaccurate, the system selects multiple regions using a greedy algorithm to provide highly meaningful information at the given amount of computing resources. The second strategy is to distribute application workload over computing resources that are placed at different locations and various levels of network hierarchy. To support this strategy, the framework provides hierarchical communication primitives and a decentralized resource discovery protocol that allow scalable and highly adaptive load balancing over space and time.

CHAPTER I

INTRODUCTION

Sensors of various modalities and capabilities, especially cameras, have become ubiquitous in our environment. Technological advances and the low cost of sensor devices enable deployment of large-scale sensor networks in various places, including urban areas, highways, airports, and stadiums. Meanwhile, various analytics for drawing inferences from live sensor streams have also matured tremendously throughout the last two decades. Such technological advances in sensors and analytics have led to the emergence of a new class of applications called *situation awareness*. These applications, including intelligent surveillance, autonomous traffic monitoring, and assisted living, monitors and controls physical environments by analyzing live streams from widely deployed sensors.

Among various modalities of sensors, live video streams have great potential benefits for situation awareness since they provide abundant and detailed information of live situation using unobtrusive cameras. For example, a retail marketing application can analyze live video streams from cameras that are deployed on product shelves to provide demographic information of customers (e.g., average age and gender) to a store manager. Using the same cameras on product shelves, a surveillance application can monitor illegal behaviors while a tracking application can find lost children in a large retail store.

Despite the wide range of use cases and potential benefits, however, using camera networks for situation awareness introduces various technical problems that prohibit large-scale applications if not addressed properly. First, most video analytics are both compute- and data-intensive, requiring use of massive distributed / parallel

computing resources across various locations and different levels of network hierarchy (e.g., computing resources in an access network and those in a data center). Secondly, applications have latency-sensitive quality of service, making it necessary to generate actionable knowledge (e.g., unauthorized access to a control room) with low latency. Lastly, situation awareness on camera networks involves highly dynamic workloads depending on the real-world situations (e.g., number of people accessing a certain area), demanding a highly adaptive system that takes into account such dynamics of the real world.

To address these challenges and enable various situation awareness applications on camera networks, this dissertation presents a distributed framework that provides a programming model and runtime optimizations. The programming model of the framework allows domain experts to easily write large-scale distributed applications by providing high-level, domain-specific programming abstractions. Based on the programming model, the runtime system performs opportunistic event processing and workload distribution over space and time to achieve low-latency processing and dynamic workload handling.

1.1 Problem Statement

Developing situation awareness applications on camera networks involves various technical challenges that prohibit domain experts from writing large-scale applications. One of the biggest challenges is to orchestrate large numbers of live video streams and computing resources to handle both comptue- and data-intensive workloads of applications. In particular, the application developers are experts in their own domain (e.g., computer vision algorithms) but are not necessarily experts in dealing with myriads of details associated with large-scale distributed systems, such as communication and synchronization. Therefore, developing applications based on thousands of live streams and computing resources is a daunting task for domain experts.

Another challenge in developing situation awareness on camera networks is meeting low-latency requirements. In most applications, processing live streams and generating actionable knowledge with low latency is critical to allow timely reaction to certain situations. For example, a traffic monitoring application should immediately notify the detection of a stolen car to a nearby police officer to allow the officer to stop the car. Similarly, a lost children finder should give immediate alarm to parents once faces of children are detected from cameras. Meeting such low-latency requirements needs taking into account both computing and networking latencies with heterogeneous resources that are widely distributed.

The last challenge is to deal with dynamic workloads of large-scale camera networks. Imagine an airport security system that performs real-time target tracking to prevent potential threats in an airport. One way of ensuring live video processing is to put enough system resources on each smart camera based on the estimated workloads. However, this approach is not suitable for highly dynamic places such as an airport since workload for target tracking in boarding areas keeps changing over time. The workload of video analytics also varies over spaces since some cameras have much more traffic (e.g., those near a security gate) than others. To deal with such real-world dynamics, a runtime system must provide highly adaptive load balancing across distributed computing resources as well as automatic adjustment of application-level fidelity to reduce the workload at the cost of less accuracy.

Because of these technical problems, developing a situation awareness application on a large-scale camera network is a daunting task to a domain expert. To enable various large-scale situation awareness applications, it is necessary to have a proper system support that shields domain experts from programming complexity and performance problems. The problem being addressed in this dissertation can be stated as the following: *What system supports do we need to facilitate the development of situation awareness applications on large-scale camera networks?*

1.2 Thesis Statement

We can enable situation awareness on camera networks by providing a distributed framework that supports a scalable programming model and a runtime system that processes events opportunistically and distributes the workload across space and time.

1.3 Contribution

This dissertation makes two main contributions. First, it proposes high-level programming abstractions that ensure scalable design of situation awareness applications, while allowing domain experts to focus on application-specific analytics. The programming abstractions expose inherent parallelism of detection, tracking, recognition, and aggregation to the runtime system, allowing the runtime system to ensure scalability and end-to-end latency requirements using parallel / distributed computing resources.

Second, it presents runtime mechanisms for low-latency stream processing and dynamic workload handling. In particular, the opportunistic event processing mechanism allows just-in-time situational information to mobile users by processing events ahead in time. To handle dynamic workload, the runtime system distributes application workloads over space and time using widely distributed computing resources at different levels of network hierarchy.

Overall, the distributed framework proposed in this dissertation fills the gap between sensing / computing resources and application logic of situation awareness, by providing a scalable programming model and a runtime system that ensures low-latency live stream processing and dynamic workload handling.

1.4 Roadmap

Chapter 2 presents related work to this dissertation. Chapter 3 and Chapter 4 explain domain-specific programming abstractions that ensure scalability of applications at two different levels of application logic: multi-camera target tracking and spatio-temporal analysis. Chapter 5 and Chapter 6 provide runtime mechanisms that support low-latency processing and dynamic workload handling. Chapter 7 discusses insights from this work and future research directions. Finally, Chapter 8 concludes this thesis dissertation.

CHAPTER II

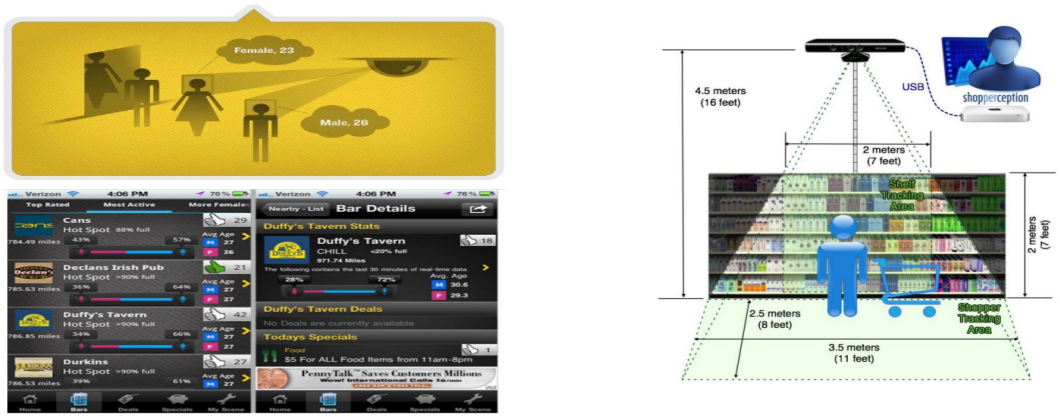
BACKGROUND AND RELATED WORK

This chapter provides application context including various use cases of situation awareness on camera networks. We also explore related work to this dissertation, including programming models in other domains and distributed systems for stream processing.

2.1 Application Context

The conventional approach to situation awareness on camera networks has required direct human involvement, either passively watching video screens or searching through recorded videos to find important events. For instance, police officers often have to manually find important evidences from video archives after crimes occurred. Although cameras are increasingly deployed over wide areas, such conventional approaches do not scale well because of the cognitive overhead of manual monitoring that causes false positives and false negatives [37].

To solve the problem, various situation awareness applications are developed to generate actionable knowledge from live videos with little to no human involvement. Smart surveillance applications are canonical examples of situation awareness on camera networks. The IBM Smart Surveillance System [21] (S3) provides extensible modules for video analytics called Smart Surveillance Engine (SSE). The SSE makes currently deployed surveillance systems "smart" based on computer vision algorithms including object detection, tracking, and classification. Using SSE, the system generates real-time alerts as well as a set of xml documents that record detailed activities within camera views.



(a) Scenetap (b) Shopperception

Figure 1: Mobile Applications using Situation Awareness on Camera Networks

Another example of situation awareness on camera networks is a real-time marketing application. Recently, Walmart deployed a video-based market analysis application called Shopperception [14] in their retail stores. Using 3D cameras, Shopperception analyzes detailed behaviors of customers, such as which products they are looking at or which products they picked up from shelves. Based on the analysis, a retail store manager can see a heat map that visualizes what products draw more attention from customers. The application also enables a customized deal for a customer who just picked up a certain product.

Various mobile applications can also provide interesting events to mobile users by analyzing live video streams. For example, a mobile application called SceneTap [74] provides age, crowed density, and gender ratio at local bars and restaurants by analyzing live video streams from unobtrusive cameras. To protect privacy, these applications can filter out privacy-sensitive information such as exact identities of individuals while providing abstract view of places using widely deployed cameras including mobile phones.

With increasing number of cameras in our environment, including mobile phones, smart vehicles, and surveillance cameras, situation awareness on camera networks will

have even greater potential in various domains. However, most of the existing applications are developed based on customized sensing and computing infrastructures, making it hard to share those resources across different applications. Our vision is to allow various applications to run on shared sensing / computing infrastructures by providing a distributed software framework. Once an application developer provides application logic through our programming model, the runtime system executes the application on shared infrastructures while performing various optimization across different applications.

2.2 Related Work

2.2.1 Stream-oriented Programming Models

Stream-oriented programming models [23, 60, 68] provide high-level programming abstractions for stream processing applications. Using their programming abstractions, application developers do not need to worry about low-level problems in distributed systems such as communication and synchronization. Instead, they can focus on writing application logic using a stream graph with computation vertices and communication edges. Once a stream graphs is provided, the runtime system of a stream-oriented programming model executes an application on distributed computing resources and performs various optimization to improve performance and resource utilization.

Although these programming models simplify developing stream processing applications, they do not provide the right level of programming abstraction for highly dynamic situation awareness applications. In particular, they require a complete stream graph for each application, which is tricky when developing a situation awareness application based on the large number of dynamic stream sources (e.g., mobile devices) and computation modules (e.g., a target tracking module). They also do not support exploiting the domain-specific parallelism of detection, tracking, recognition and aggregation while they are prevalent in situation awareness applications. Lastly,

application logic in different stream stages can only communication through stream channels, which makes it difficult to share real-time data among computation modules (e.g, sharing current position of individual targets among detection and tracking modules to avoid redundant detection).

2.2.2 Programming Models for Sensor Networks

Many programming models have been proposed to address various issues of developing sensor network applications. Abstract Regions [72] provides interfaces for identifying neighboring nodes, sharing data among neighbors, and performing reductions on shared variables to support efficient aggregation on sensor networks. EnviroSuite [50] provides an object-oriented programming model that allows programmers to think physical elements in the external environment as programming objects. Semantic Streams [73] supports declarative queries over semantic interpretations of sensor data, such as a query on a certain target instead of a specific raw sensor stream. Smart Messages [7] allows applications to execute on nodes of dynamic interests, specified by spatial properties, using an explicit lightweight migration mechanism.

Although these programming models support programming on the large number of sensor devices, they assumed structured sensor data (e.g., integer values) or event-driven sensor streams such as RFID. For situation awareness on camera networks, video analytics have to run on continuous video streams to detect high-level events while those events are inherently uncertain. Our programming models are specialized for camera networks to deal with such continuous streams and uncertain events.

2.2.3 Distributed Systems for Camera Networks

Many distributed systems have been developed to support various applications on camera networks. EasyLiving [10] provides a software architecture for smart environments including a person tracking system that identifies different people over multiple rooms using color-based features. IBM Smart Surveillance System [21] provides

extensible modules for object detection, tracking and classification, as well as a middleware for event and video management. ASAP [66] presents a scalable distributed architecture for a multi-modal camera network that provides selective attention on important video streams using priority cues. It also supports redirection of streams to handle bursty workload from different locations. CITRIC [11] presents a low-bandwidth wireless camera platform and a backend system that supports distributed image compression, target tracking, and camera localization. SensEye [46] presents a multi-tier network of heterogeneous wireless nodes and cameras, which achieves both low latency and energy efficiency.

These systems are complementary to our framework since they address different pain points of developing distributed applications on camera networks. Compared to these existing systems, our novelty is to provide a distributed programming framework that exploits inherent parallelism of situation awareness on camera networks.

2.2.4 Complex Event Processing Systems

Many Complex Event Processing (CEP) systems [59, 47, 1, 15, 42] provide continuous query interfaces and optimization mechanisms for detection of interesting patterns from sensor data. To reduce latency for processing events, some CEP systems exploit parallelism [16, 31] while others support adaptive placement of operators [63, 58].

While these systems are mainly designed to support efficient queries on structured data such as strings and integers, the main goal of our framework is to generate such structured actionable knowledge from unstructured video streams with low latency. To support higher-level queries on actionable knowledge, those CEP systems can be combined with our framework.

2.2.5 Spatio-temporal Databases

Research in spatio-temporal databases has developed various representations of spatio-temporal objects and methods for querying and storing spatio-temporal objects [18,

57, 26]. These spatio-temporal databases are complementary to our work since they can serve as a spatio-temporal event storage module in the framework.

Predictive query handling on spatio-temporal databases [38, 30] allows answering queries about the future locations of mobile objects, e.g., ten nearest neighbors after five minutes, by predicting locations of mobile objects. Hendawi et al. [30] proposed precomputing query results to improve scalability and reduce the latency of query handling. In contrast to these work, our opportunistic event processing mechanism delivers just-in-time situational information for customized queries for individual mobile users, where each query is about the recent state of the current location.

2.2.6 Computing Platforms for Situation Awareness Applications

Cloud offers elastic computing resources for stream processing, allowing applications to deal with real-world dynamics such as workload fluctuations and resource failures. TimeStream [61] provides a mechanism called resilient substitution to support dynamic reconfiguration at runtime in response to server failures and load fluctuations. MillWheel [2] provides the notion of logical time to help writing time-based aggregations while supporting fault-tolerant stream processing at Internet scale. S4 [53] supports scalable stream processing on keyed data using *emit* and *publish* operations. While these systems provide highly scalable solutions for stream processing, they require input streams to flow through Internet to reach cloud computing resources. As Clinch et al. [12] showed, such WAN latencies between the cloud and edge devices can be harmful for latency-sensitive applications, such as interactive applications and situation awareness applications.

Two systems that can provide computing resources near the edge of the network are MediaBroker [48] and Cloudlets [64]. While they support live sensor stream analysis and interactive applications respectively, neither currently supports widely distributed situation awareness applications. Content Distribution Networks (CDNs),

on the other hand, push resources to the edge of the network [17] over wide areas. However, CDNs are limited to satisfying requests for content, rather than supporting generic application logic. Recent advances in software–defined networking (SDN) [43] allow programming in-the-middle network resources. However, these approaches only allow injecting routing logic into network elements, not generic application logic.

To support latency-sensitive, large-scale situation awareness applications, we propose a fog-based execution environment called *Mobile Fog* in Chapter 6. Mobile Fog is designed based on a new computing paradigm, called *fog computing*, that is proposed by Bonomi et al. [6]. While they defined the characteristics of the fog computing and showed potential benefits of the fog through various use cases, managing highly dynamic fog computing resources remains a problem. Mobile Fog solves the problem by providing hierarchical communication primitives and automatic resource adaptation.

2.2.7 Video Analytics for Situation Awareness

Various video analytics are developed for situation awareness applications through the last two decades. The state of the art video analytics include background subtraction [67, 20], feature-based object detection [70], target tracking [29, 77], recognizing people based on biometric information such as face [78] and gait signatures [49], and understanding human motion and activities [22, 5].

While these video analytics serve enabling technologies for situation awareness on camera networks, they focus on the accuracy of applications rather than scalability and performance issues that are critical in large-scale scenarios. To support these analytics in large-scale environments, our framework allows domain experts to easily plug in their video analytics using domain-specific handlers. At runtime, our system automatically invokes those handlers using distributed system resources to ensure scalability and low-latency stream processing.

CHAPTER III

TARGET CONTAINER

This chapter discusses the first part of our programming model, Target Container [35]. Target Container serves a parallel programming abstraction for developing complex situation awareness applications that track multiple targets in large-scale camera networks. The key insight of this work is to elevate a physical target as the first class citizen, providing an intuitive programming abstraction for domain experts while enabling efficient resource management for different targets. In the rest of this chapter, we use a surveillance application as a canonical example of multi-camera target tracking applications that are supported by TC programming model.

3.1 Application Logic

Let us first understand the general logic of surveillance applications. In general, surveillance applications have two key functions: detection and tracking. Detection primarily focuses on finding an event that may be of interest to a surveillance application. For example, there are many control rooms in an airport that people are not allowed to access. If an unauthorized person tries to access a control room, an automated surveillance system should capture the event among thousands of normal activities in the airport. Once an event is detected, the automated surveillance system should keep track of the target that triggered the event. While tracking the target across multiple cameras, the surveillance system should provide all relevant information of the target, including the current location and multiple views of the target, helping a security team react to the threatening target.

To track a target across multiple cameras, computer vision researchers have developed algorithms based on different types of features [21, 28, 51]. These techniques

essentially compare various features of targets to identify the same target in multiple video streams. If a target simultaneously appears in the field of view (FOV) of multiple cameras, trackers following the target on each of the different camera streams need to work together to build a composite feature of the target.

The general application logic represents the inherent parallel/distributed nature of surveillance applications. Each detector is a per camera computation that exhibits a massive data-parallelism since there is no data dependency among detectors working on different camera streams. Each tracker is a per target computation that can run simultaneously on each target. Comparison of two targets in different camera streams can run in parallel with comparison of other pairs of targets.

There also exist complex data sharing and communication patterns among different instances of detectors and trackers. For example, it is necessary to compare up-to-date target data generated by trackers and object detection results generated by a detector to find new targets while avoiding redundant detection. If a detected object has similar features (e.g., location, color, etc.) with an existing target, the two objects may be the same.

3.2 Conventional Approaches to Developing Surveillance Applications

In this section, we motivate our approach by presenting limitations in developing surveillance applications based on existing programming models. We consider a couple of different approaches (namely, *thread-based* and *stream-oriented*) and identify the shortcomings of those approaches for large-scale surveillance applications before presenting the TC programming model.

3.2.1 Thread-based Programming Model

The lowest-level approach to building surveillance systems is to have the application developer handle all aspects of the system, including traditional systems aspects, such

as resource management, and more application-specific aspects, such as mapping targets to cameras. Under this model, a developer wishing to exploit the natural parallelism of the problem has to manage the concurrently executing threads over large number of computing nodes. This approach allows the developer to optimize the computational resources most effectively since he/she has complete control over the runtime system and the application logic.

However, carefully managing computational resources for multiple targets and cameras is a daunting responsibility for surveillance application programmer. For example, the shared data structure between detectors and trackers ensuring target uniqueness should be carefully synchronized to achieve the most efficient parallel implementation. Multiple trackers operating on different video streams may also need to share data structures when they are monitoring the same target. These complex patterns of data communication and synchronization place an unnecessary burden on an application developer, which is exacerbated by the need to scale the system to hundreds or even thousands of cameras and targets in a large-scale deployment (e.g., airports, cities).

3.2.2 Stream-oriented Programming Model

Another approach to developing a surveillance application is to use a stream-oriented programming model [24, 53, 68, 60] as a high-level abstraction. Under this model, the programmer does not need to deal with low-level system issues such as communication and synchronization. Rather, she can focus on writing an application as a stream graph consisting of computation vertices and communication edges. Once a programmer provides necessary information including a stream graph, the underlying stream processing system manages the computational resources to execute the stream graph over multiple nodes. Various optimizations are applied at the system level, shielding the programmers from having to consider performance issues.

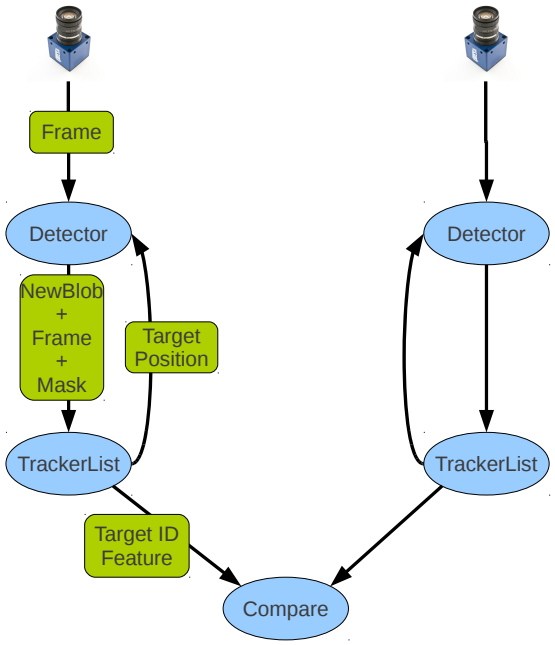


Figure 2: Target Tracking based on Stream-oriented Models

Figure 2 illustrates our attempt to implement a target tracking application using IBM System S [23], one of the representative off-the-shelf stream processing engines. In the application, a detector processes each frame from a camera, and produces a data item containing three different information: newly detected blobs, an original camera frame, and a foreground mask. A second stream stage, trackerlist, maintains a list of trackers following different targets within a camera stream. It internally creates a new tracker if newly detected blobs are received by a detector. Each tracker in a trackerlist uses an original camera frame and a foreground mask to update each target's blob position. The updated blob position will be sent to a detector, to prevent redundant detection of the target.

Using the surveillance application based on IBM System S, we identified several critical drawbacks of stream-oriented programming models for developing large-scale surveillance applications. First, stream-oriented programming models require a complete stream graph for each application. Even if the number of cameras is static

and the locations of cameras do not change, building a large stream graph based on camera proximity can be a non-trivial task when it comes to thousands of cameras. Second, the stream-oriented approach is not well-suited for exploiting the inherent parallelism of target tracking. For example, when a new target is detected, a new instance of tracker should be created to track the target. There is an opportunity to execute multiple trackers concurrently if the computing infrastructure supports hardware parallelism. To exploit such target tracking parallelism, it is necessary to create a new stream stage to track the new target. However, dynamically creating a new stream stage is not supported by System S and therefore a single stream stage (the stage labeled trackerlist in Figure 2), should execute multiple trackers internally. This makes a significant load imbalance of different trackerlists, as well as low target tracking performance due to the sequential execution of trackers. Lastly, stream stages can only communicate through stream channel, which prohibits arbitrary real-time data sharing among different computation modules. As shown in Figure 2, a programmer has to explicitly connect stream stages through stream channels and deal with communication latency under conditions of infrastructure overload.

3.3 Programming Abstraction

Based on the limitations of existing programming models described in the previous section, we present the design of our new programming model, Target Container. TC programming model is designed for domain experts who want to rapidly develop large-scale surveillance applications. In principle, the model generalizes to dealing with heterogeneous sensors (cameras, RFID readers, microphones, etc.). However, for the sake of clarity of the exposition, we adhere to cameras as the only sensors in this paper.

We assume that the physical deployment topology of the camera network is known

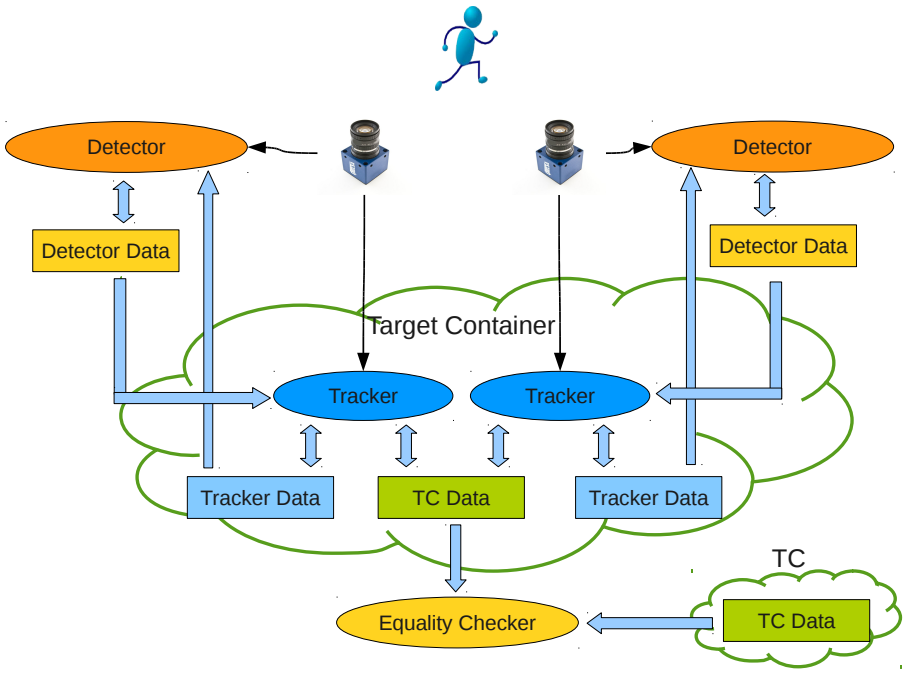


Figure 3: Situation Awareness Application using Target Container

to the execution framework of the TC system. With Target Container, the connections between the cameras and computing resources, as well as local communication among nearby smart cameras are orchestrated under the hood. This design decision has the downside that it precludes an application developer from directly configuring inter-camera communication. However, we believe that domain experts (e.g., vision researchers) would much rather delegate the orchestration of such communication chores to the system, especially when it comes to thousands of distributed cameras. Consequently, an application developer only needs to deal with the algorithmic aspect of target tracking based on the TC abstraction.

3.3.1 Handlers and API

The intuition behind the TC programming model is quite simple and straightforward. Figure 3 shows the conceptual picture of how a surveillance application will be structured using the new programming model and Table 1 summarizes APIs provided by the TC system. The application is written as a collection of handlers. There is a

Table 1: Target Container API

Interface	**Description**
API	Description
TC_create_target()	It creates a TC and associates it with the new target. This is called by a detector. It also associates a tracker within this TC for this new target, which analyzes the same camera stream as the detector.
TC_stop_track()	When a target disappears from a camera's FOV, tracker makes this interface call to prevent further execution of itself.
TC_get_priority()	Get a priority of a TC.
TC_set_priority()	Set a priority of a TC.
TC_update_detector_data()	This will be used by detector for updates to per detector data structures.
TC_read_detector_data()	This will be used by detector/tracker for read access to per detector data structures.
TC_update_tracker_data()	This will be used by tracker for updates to per tracker data structures.
TC_read_tracker_data()	This will be used by detector/tracker for read access to per tracker data structures.
TC_update_TC_data()	This will be used by tracker for updates to per TC data structures.
TC_read_TC_data()	This will be used by detector/tracker for read access to per TC data structures.

detector handler associated with each camera stream. The role of the detector handler is to analyze each camera image it receives to detect any new target that is not already known to the surveillance system. The detector creates a *target container* for each new target it identifies in a camera frame by calling *TC_create_target* with initial tracker and TC data.

In the simple case, where a target is observed in only one camera, the target container contains a single *tracker* handler, which receives images from the camera and updates the target information on every frame arrival[1]. However, due to overlapping

[1]Since the physical deployment topology of the camera network is available to the execution framework of the TC system, the specific camera stream is implicitly delivered to the newly spawned tracker by the TC system.

fields of view, a target may appear in multiple cameras. Thus, in the general case, a target container may contain multiple trackers following a target observed by multiple cameras. A tracker can call *TC_stop_track* to notify the TC system that this tracker need not be scheduled anymore; it would do that upon realizing that the target it is tracking is leaving the camera's field of view.

In addition to detectors (one for each sensor stream) and trackers (one per target per sensor stream associated with this target), the application must provide additional handlers to the TC system for the purposes of merging TCs as explained below. Upon detection of a new target in its field of view, a detector would create a new target container. However, it is possible that this is not a new target but simply an already identified target that happened to move into the field of view of this camera. To address this situation, the application would also provide a handler for *equality checking* of two targets. Upon establishing the equality of two targets, the associated containers will be merged to encompass the two trackers (see Target Container in Figure 3). The application would provide a *merger* handler to accomplish this merging of two targets by combining two application-specific target data structures (TC data) into one. Incidentally, the application may also choose to merge two distinct targets into a single one (for example, consider a potential threat situation when two cohorts join together and walk in unison in an airport).

3.3.2 Data Structures

As shown in Figure 3, there are three categories of data with different sharing properties and life cycles. Detector data is the result of processing the per-stream input that is associated with a detector. The data can be used to maintain detector context such as detection history and average motion level in the camera's field of view, which are potentially useful for applications using per camera information. The detector data is potentially shared by the detector and the trackers spawned thereof. The trackers

spawned by the detector as a result of blob detection may need to inspect this detector data. The tracker data maintains the tracking context for each tracker. The detector may inspect this data to ensure target uniqueness. TC data represents a target. It is the composite of the tracking results of all the trackers within a single TC. The equality checking handler inspects the TC data to see if two TCs pertain to the same target, and if so calls the merger handler to merge the two TCs and create one composite TC data. Building such a composite data structure is in the purview of the domain expert.

The TC programming model allows dynamic data sharing between cameras and server nodes. This flexibility means that programmers do not have to statically set up the data communication among the camera nodes at application development time. Dynamically, the required communication topology among the camera nodes and the cluster nodes can be set up depending on the current needs of the application. Further, as described above, different handlers need access to the different categories of shared data at different points of time in their respective execution. Thus, providing access to shared data is a basic requirement handled in the TC system.

While all three categories of data are shared, the locality and degree of sharing for these three categories can be vastly different. For example, the tracker data is unique to a specific tracker and at most shared with the detector that spawned it. On the other hand, the TC data may be shared by multiple trackers potentially spanning multiple computational nodes if an object is in the FOV of several cameras. The detector data is also shared among all the trackers that are working off a specific stream and the detector associated with that stream. This is the reason our API (see Table 1) includes six different access calls for these three categories of shared data.

3.3.3 Parallel Execution of Handlers

When programming a target tracking application, the developer has to be aware of the fact that the handlers may be executed concurrently. Therefore, the handlers should be written as sequential code with no side effects to shared data structures to avoid explicit application-level synchronization. TC programming model does not sandbox handlers to allow application developer to use optimized handlers written in low-level programming languages such as C and C++. Data sharing between different handlers are only allowed through TC API calls (shown in Table 1), which subsume data access with synchronization guarantees.

3.3.4 Merging Target Containers

To seamlessly merge two TCs into one while tracking targets in real time, TC system periodically calls equality checker on candidates for merge operation. To avoid side effects while merging, TC system ensures that none of those trackers in the two TCs are running. After merge, one of the two TCs is eliminated, while the other TC becomes the union of the two previous TCs. Execution of the equality checker on different pairs of TCs can be done in parallel since it does not update any TC data. Similarly, merger operations can go on in parallel so long as TCs involved in the parallel merges are all distinct.

TC system may use camera topology information for efficient merge operations. For example, if many targets are being tracked in a large scale camera network, only those targets in nearby cameras should be compared and merged to reduce the performance overhead of real-time situation awareness applications. Although discovery of camera connectivity in large scale is very important issue, it is outside the scope of this work.

3.4 System Implementation

Our current implementation of TC system is in C++ and uses the Boost [65] thread library and OpenMPI [25] library. The TC runtime system implements each detector handler as a dedicated thread running on a cluster node. The TC system creates a pool of worker threads in each cluster node for scheduling the execution of the tracker handlers. The number of worker threads are carefully selected to maximize CPU utilization while minimizing context switching overhead.

The TC paradigm addresses parallelism at a different level in comparison to potentially parallel implementations of the OpenCV [9] primitives. For example, OpenCV provides options to use the Intel TBB [62] library and the CUDA [54] programming primitives to exploit data-parallelism and speed up specific vision tasks. The TC programming model deals with parallelism at a coarser level, namely, multiple cameras and multiple targets. This is why TC uses OpenMPI, which supports multi-core/multi-node environments. Using the TC paradigm does not preclude the use of local optimizations for vision tasks *a la* OpenCV. It is perfectly reasonable for the application-specific handlers (trackers, detectors) to use such optimizations for their respective vision tasks.

In the prototype TC system, each cluster node has a TC scheduler that executes trackers running on a camera stream. The TC scheduler performs priority-aware resource scheduling to ensure real-time tracking of important targets. TC data sharing between trackers is achieved via MPI communication. Handler migration across different computing nodes in the case of node failure or load imbalance is our future work, since the programming model provides a great degree of control to the underlying system.

Since the prototype TC system is designed for real-time tracking, it does not maintain a queue for past camera frames; it overwrites the previous frame in a frame buffer if a new frame arrives. To avoid any side effects from overwriting frames,

each tracker has its own duplicated frame for use by the tracking code. This ensures trackers always work with the most up-to-date camera frame. In this design, however, trackers can skip several frames if the system is overloaded. This is a trade-off between accuracy and latency; maintaining a frame queue will ensure that trackers process all the camera frames but such a design choice will introduce high latency for event detection under overloaded conditions.

3.5 Prototype Surveillance Application using Target Container

To prove the simplicity of developing a surveillance application using TC programming model, we developed an example application using existing computing vision algorithms. In the example application, the detector handler uses blob entrance detection algorithm as shown in Figure 38. The detector discovers a new object within a camera's FOV by comparing the new_blob_list (obtained from the blob entrance detection algorithm) to the old_blob_list (from existing trackers running on the camera). The old_blob_list represents up-to-date position of existing targets within a single camera's FOV since each tracker is asynchronously updating its position. If the detector finds a new blob that does not overlap with any other existing targets, it creates a new TC by calling *TC_create_target* with initial data associated with the new tracker and TC.

Once created, a tracker follows a target within a single camera's FOV using a color-based tracking algorithm as described in Figure 39[2]. In the tracker, *color_track* function computes the new position of the target using a blob-tracking algorithm from the OpenCV library. If the target is no longer in the image as determined by the *is_out_of_FOV* function (i.e., the target has left the FOV of the camera), the tracker

[2]The suggested implementation of the tracker and detector are for illustration purposes on the ease of development of a complex application using the TC model. As such, the choice of the algorithms for tracking, detection, equality checking, and merging is in the purview of the domain expert.

requests the system to stop scheduling itself by calling *TC_stop_track*. While tracking, application level target priority may change over time, depending on a target's behavior or location. Using *TC_get_priority* and *TC_set_priority*, an application can notify a target's priority to the TC system. This is vital for priority-aware resource management of the TC system. Tracker in this application also updates TC data if it finds color histogram of the target has been changed more than a threshold. Updating TC data has performance implications due to data sharing among multiple nodes, although the actual cost depends on the mapping of handlers to physical nodes. Because of this, updating TC data sparingly is a good idea.

Figure 40 illustrates examples of an equality checker and a merger. An equality checker compares two color histograms and returns TRUE if the similarity metric exceeds a certain threshold. Details such as setting the threshold value are in the purview of the domain expert and will depend on a number of environmental conditions (e.g., level of illumination). Such details are outside the scope of the TC programming model. The *compare_hist* function is implemented based on histogram comparing function from OpenCV. Merger simply averages two color histograms and assigns the result histogram to a newly merged TC data.

The above examples are overly simplified illustration of the application logic for demonstrating the use of the TC system API. A sophisticated application may contain much more information than a blob position and a color histogram to represent the target data structure. For example, the target data structure may contain a set of color histograms and trajectories for different camera views of the same target. The equality checker and merger handler will be correspondingly more sophisticated, effectively comparing and merging two sets of color histograms and trajectories.

3.6 System Evaluation

This section evaluates the scalability of the TC system compared to a conventional video surveillance system. To do so, we performed three different experiments with both systems in various practical situations. These experiments serve to answer the following questions:

- How do the conventional and TC systems scale as the number of targets increases?
- What are the benefits due to TC target prioritization, under conditions of performance saturation?

For the purposes of this evaluation, we use the metric *target throughput*. The target throughput is the number of target completion during an inter-frame interval, where target completion means executing all the trackers associated with a target. The target throughput percentage is the ratio of target completion to targets. Assuming the frame rate of a video camera stream is fixed and number of targets is constant, the target throughput will remain steady if the system is underloaded. This is because the system has enough resources to process all the targets during each inter-frame interval. However, when the system is overloaded, i.e., the system does not have enough resources to process all the targets during an inter-frame interval, some targets may not be fully executed (i.e., some trackers following a target may not have been executed) before the next frame arrives. In this case, the overall target throughput will decrease as new targets are added since each target handler will have less of a chance to be executed at each frame arrives.

3.6.1 Experimental Setup

The experiments are conducted on Linux kernel 2.6.32 with an Intel Q6600 2.40GHz CPU with four cores. To reduce the noise in the experiments, we dedicated one core to

run all the detector handlers for all the cameras, as well as the internal threads needed by the TC runtime system (e.g., to carry out merger operations). Incidentally, the TC runtime system uses a daemon thread to periodically invoke the equality checker handler provided by the application to determine if TCs have to be merged. The remaining three cores are dedicated for running the tracker handlers using the worker thread pool that we described in the previous section. To ensure that the first core is not overloaded, we emulate physical cameras by replaying video files with a fixed frame rate.

In our setup, a video camera emulator processes each video frame image to differentiate between foreground and background image. This is the first processing step in a video surveillance pipeline and would typically run on smart cameras under a real-world deployment scenario. In our setting, it takes about 150 ms to process foreground detection for one 800x600 image. A blob entrance detector (which would also typically run on smart cameras) identifies new objects in each video frame, and takes 20ms on an average. To safely provide a fixed video frame rate for our experiments, we set the frame rate of the video camera emulator to five frames per second. This rate provides enough time to process foreground detection and blob entrance detection, in our experimental testbed. We use a color tracker based on the *mean shift* algorithm [13] to track targets. The color tracker implementation from the OpenCV library takes 30-50ms to track a target in our setting. Assuming a 30x30 pixel object within a 800x600 pixel video, we set the overhead for tracking a target to be 30ms. To provide a fair comparison, these settings are fixed for all of our experiments.

3.6.2 Target Throughput Scaling

Base Scenario. In this experiment, we measure the effects of increasing load on the surveillance system under test. Figure 4 shows the relationship of CPU utilization and average target throughput. Until the CPU utilization increases to over 90%,

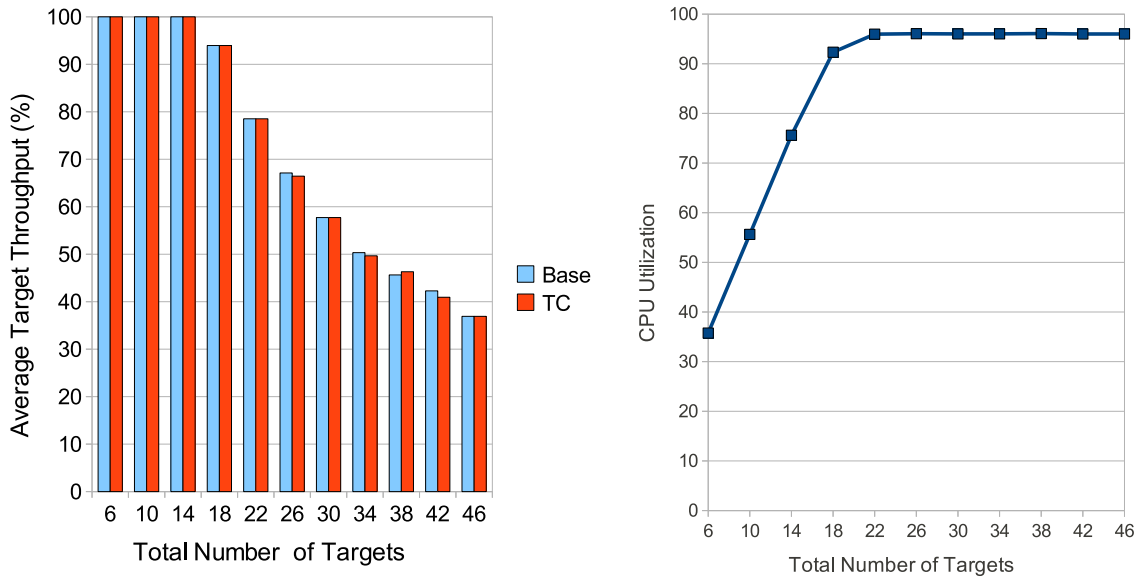


Figure 4: Effects of System Load on Target Handler Throughput

the system is underloaded and the average target throughput does not change even though new targets are added. However, as the number of targets increase from 14 to 18, the average target throughput starts to decrease due to system overload. Note that TC system and the baseline system[3] do not show any difference in this experiment since every target has only one tracker (i.e., is in the FOV of exactly one camera).

Single Resource Hungry Target. In this experiment, we assume a situation where one target is in the FOV of multiple cameras while other targets are in the FOV of a single camera. Although it may not be realistic that a single target is observed by tens of cameras at the same time, we used this scenario to emulate a situation that tracking a certain target consumes a large amount of system resources. To evaluate relative scalability between the TC and the baseline systems, we measure the average target throughput of TC and the baseline systems as the number of trackers for the first target increases. Note that the number of targets do not change while the number

[3]We refer to the conventional thread-based implementation as the "baseline" system.

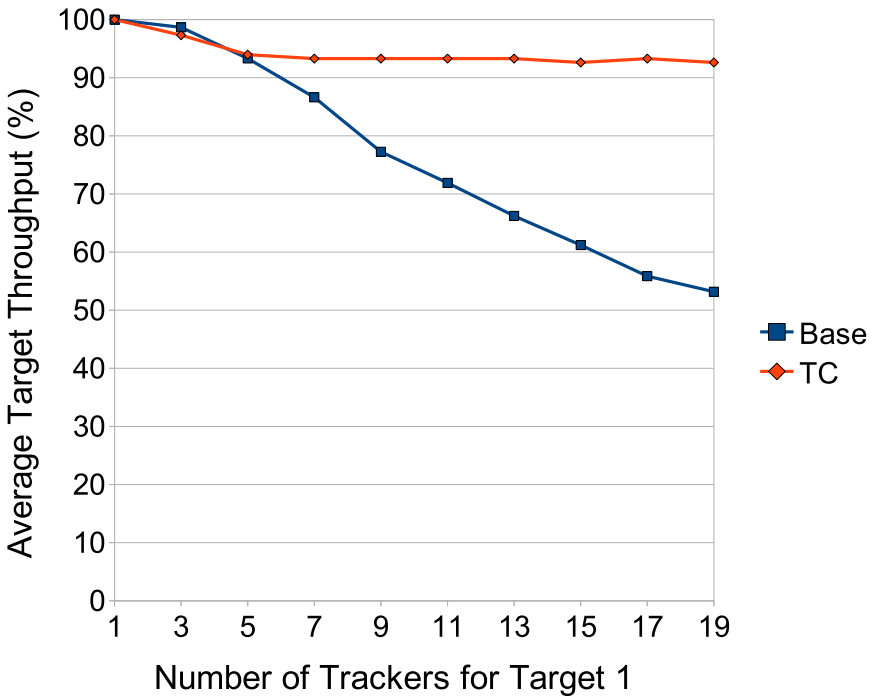


Figure 5: Effects of Single Resource Hungry Target

of trackers increases. Figure 5 shows the average target throughput with 14 targets for both TC and the baseline systems.

As depicted in Figure 4, both systems can run 14 trackers without throughput saturation. However, the average target throughput of the baseline system starts to degrade as soon as additional trackers are added to the first target, since the system is overloaded beyond 14 trackers. However, the average target throughput of the TC system does not degrade much, because only the first target's throughput will be affected when more trackers are added to it. As shown in Figure 5, the target throughput of the baseline system drops to nearly 50% of the original target throughput measured in an underloaded condition, because the number of trackers are increased. On the other hand, the average target throughput of the TC system remains well over 90% because the number of targets remains the same. This result demonstrates that the TC system fairly provides all targets of equal priority equitable amount of computational resources when the system is overloaded.

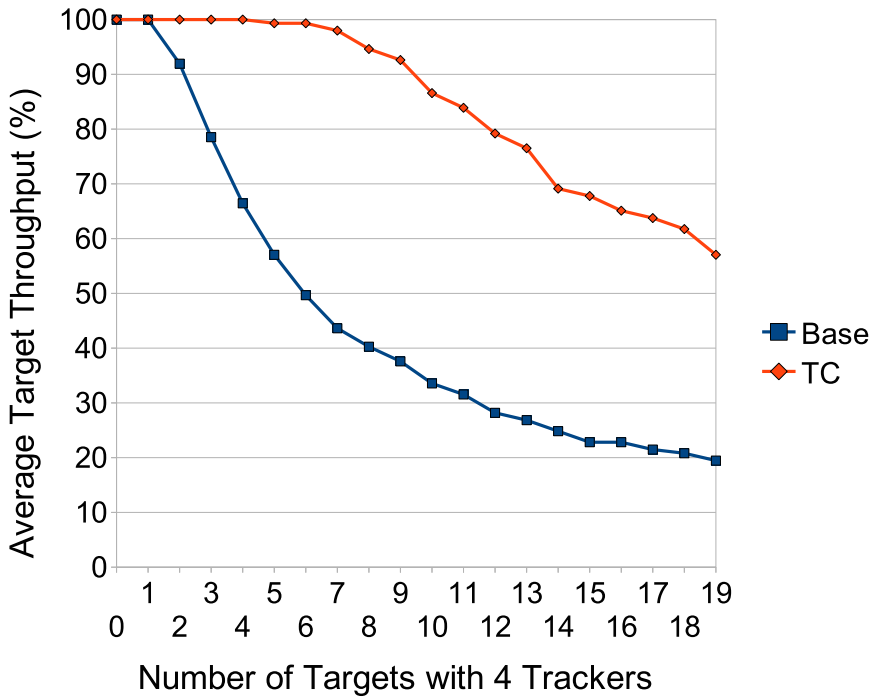


Figure 6: Effects of Many Resource Hungry Targets

Many Resource Hungry Targets. This experiment represents a situation where the number of resource hungry targets that are in the FOV of multiple cameras is increased. In this experiment, ten targets with a single tracker are initially running and we gradually increase the number of targets with four trackers. Figure 6 shows the different scalability of both systems in terms of the average target throughput.

Initially, both TC and baseline systems show 100% of average target throughput since they are not overloaded yet. However, the baseline system rapidly degrades once resource hungry targets are added and the number of trackers exceeds 14 trackers. Although TC system also starts to degrade when more resource hungry targets are added, it degrades more gracefully. This is because the number of trackers in the baseline system increases faster than the number of targets in the TC system, allowing the TC system provides higher average target throughput as it provides equal resources to targets, not trackres.

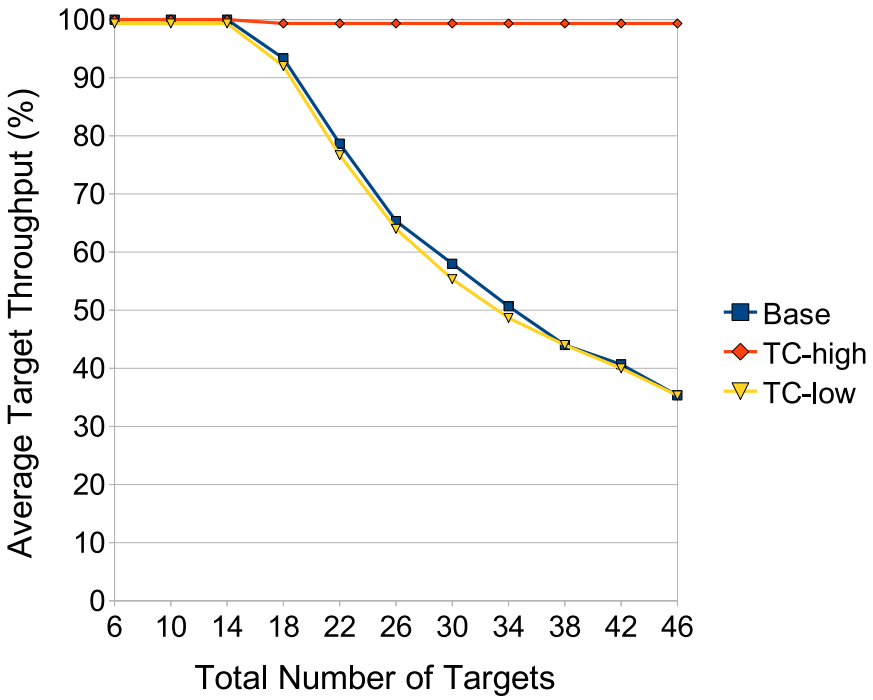


Figure 7: Effects of Target Prioritization

3.6.3 Target Prioritization

The next experiment assigns different priorities (high or low) to different targets. In this scenario, we assume there exists one target that is more important to track than the others. Regardless of system load state, the target tracking algorithm should successfully track the high-priority target. To emulate this scenario, we assign high priority to one of the targets running on the system. The rest of targets have the same low priority. Although the TC system can allocate resources based on a gradient scale of target priorities, we only choose two priorities (high and low) for the sake of simplicity. All targets have one tracker (i.e., in the FOV of a single camera) in this experiment. At initialization, six targets are being tracked under the TC and the baseline systems. Figure 7 presents the results of this experiment.

When the experiment begins, all targets are subject to the same target throughput regardless of priority since both systems are not yet saturated. Beyond 14 targets, the

systems reach saturation, and the low priority targets in the TC system experience reduced throughput. Similarly, all targets in the baseline system experience reduced throughput performance. However, the high priority target in the TC system still receives enough resources to be successfully tracked throughout the experiment. The low priority targets in the TC system have slightly lower average target throughput than the baseline system since the high priority target receives more resources than others. However, only a few selective targets will likely require a higher level of attention in most practical situations and therefore, the loss in average target throughput for low priority targets is likely to be considered an acceptable trade-off.

3.7 Conclusion

In this chapter, we have proposed *Target Container* (TC), a novel parallel programming abstraction for multi-camera target tracking applications. The TC programming model provides the following key benefits: First, programmers need not deal with low-level thread management; nor do they have to provide a complete stream graph. Building a large-scale video surveillance application boils down to writing four handlers that implements application-specific computer vision algorithms. Second, decoupling the programming model from its execution framework makes it easier to exploit domain-specific parallelism. Spawning the detectors and trackers on an actual execution platform consisting of smart cameras and cluster nodes is the responsibility of the TC runtime system. Third, the TC system subsumes the buffer management and synchronization issues associated with real-time data sharing among the different instances of detectors and trackers. Finally, the TC system allows the application programmer to specify priorities for the targets that it is currently tracking. This information is available to the TC system to orchestrate the allocation of computational resources commensurate with the priority of the targets.

CHAPTER IV

SPATIO-TEMPORAL ANALYSIS

This chapter discusses the second part of our programming model that supports spatio-temporal analysis [34, 36] on camera networks. Situation awareness applications often rely on a technique called *spatio-temporal analysis* to answer queries on occupants such as "When did person O leave zone Z?". Applications providing means to answer these queries usually employ distributed cameras and sensors of other modalities (such as audio and biometrics) to detect people in the observed system. Using these live sensor streams, applications make estimates about identities of detected people by comparing sensor data to a set of well-known identities, and gather those estimates to create a global view of the observed area.

Recently, Menon et al. [52] showed the feasibility of spatio-temporal analysis with a global state transition table. The table represents the probabilities of each occupant known to the system being in each of the observed zones. An events, which indicates that an occupant has been observed within a zone, triggers a transition from the current state to the next. Just as the global state, events are represented by probabilities rather than exact knowledge, because algorithms for signature detection and comparison are inherently inaccurate. While the global state transition table is useful for query handling, keeping a global application state at a central server imposes a significant performance bottleneck. Thousands of cameras constantly send updates to that server, drastically increasing the communication costs, and the server has to potentially perform a vast number of computations to process all the updates.

To allow domain experts to focus on algorithmic details of application logic instead of dealing with such performance problems, we propose a distributed programming

abstraction for spatio-temporal analysis. Specifically, our contribution includes 1) the design of distributed programming abstraction for spatio-temporal analysis, 2) investigation of performance bottleneck for spatio-temporal analysis on large-scale camera networks, and 3) scalable mechanisms that address computation and communication overheads of state update.

4.1 Application Logic of Spatio-temporal Analysis

In this section, we explain the general logic of spatio-temporal analysis on camera networks. Spatio-temporal analysis enables an application to answer queries referring to locality- and time-dependent information about different occupants. Common examples of spatio-temporal queries include:

"Where was person A at time T?"

"When did person B leave zone X?"

"When and where did person A and B meet for the last time?"

"Who moved from zone X to zone Y between time T1 to T2?"

To answer these queries, an application has to maintain its state which represents each occupant's location at different point of time. Figure 8 shows a general application pipeline of spatio-temporal analysis involving four steps: signature detection, event generation, state update, and query handling.

Signature detection involves video analytics to detect signatures such as faces. For example, when a person enters a zone, a face detection algorithm reports the person's face by analyzing video frames from a camera observing the zone. Note that multiple signatures can be reported from a single frame. For each frame, the face detection algorithm finds all image regions containing faces and passes them to the event generation step.

Event generation involves generating a probabilistic estimate about the identity of a detected signature. Depending on the application, different algorithms can be used

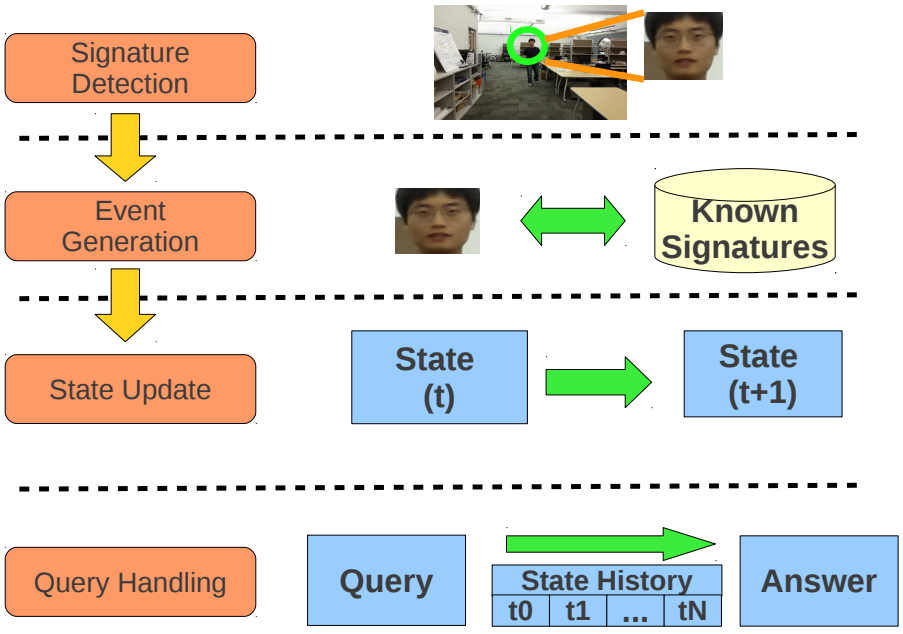


Figure 8: General Application Logic of Spatio-temporal Analysis

in this step. For example, various face recognition [78] or human gait recognition [71, 40] algorithms may be used to generate an event that includes similarities between the detected signature and known signatures.

State update maintains an application-specific state based on the observed events. The goal is to reflect in the global state the information provided by an event, e.g., that Person A was seen in Zone 2 with a probability of 0.75. The state of an application represents its knowledge about each occupant's location at a given time. A new event causes an update from one state to another, using the new information about a specific occupant's location. Different state update algorithms can be used depending on the application needs. For example, Menon et al. [52] proposed a simple state transition function that increases probabilities of an occupant being in a zone proportionally to the similarities between the detected signature and known signatures. More complex algorithms may be used, such as one taking adjacent zones and possible paths across zones into account for better accuracy.

```
Signature[] detect_signature(VideoFrame);
EventElement[numOccupants] generate_event(Signature);
StateElement[numZones] update_state(EventElement,\
    StateElement[numZones]);
```

Figure 9: Handlers for Spatio-temporal Analysis

Query handling uses the current and past application states to answer various spatio-temporal queries. Although it is an essential step for situation awareness applications, we do not consider query handling for the system design since it is not in the critical path of real-time event processing.

4.2 Programming Abstraction

Our programming abstraction for spatio-temporal analysis (STA) allows domain experts to simply provide three handlers to implement spatio-temporal analysis on large-scale camera networks. Figure 9 shows the application-specific handlers in the abstraction that are supposed to be provided by domain experts: *detect_signature*, *generate_event*, and *update_state*. Once registered, each handler is invoked automatically by the runtime system to process corresponding input data. For example, *detect_signature* handler is invoked when a new frame is available while *generate_event* is invoked when a new signature is captured. In the following subsections, we discuss detailed roles of those handlers and illustrate how they run on distributed computing resources.

4.2.1 Logical Roles of Handlers

Figure 10 shows the roles of handlers along with their input and output data. The role of *detect_signature* is to analyze each camera image it receives to detect signatures. A domain expert would code up a video analytics algorithm in this handler that detects application-specific signatures such as faces and human gaits. The handler returns an array of detected signatures where each signature is automatically tagged

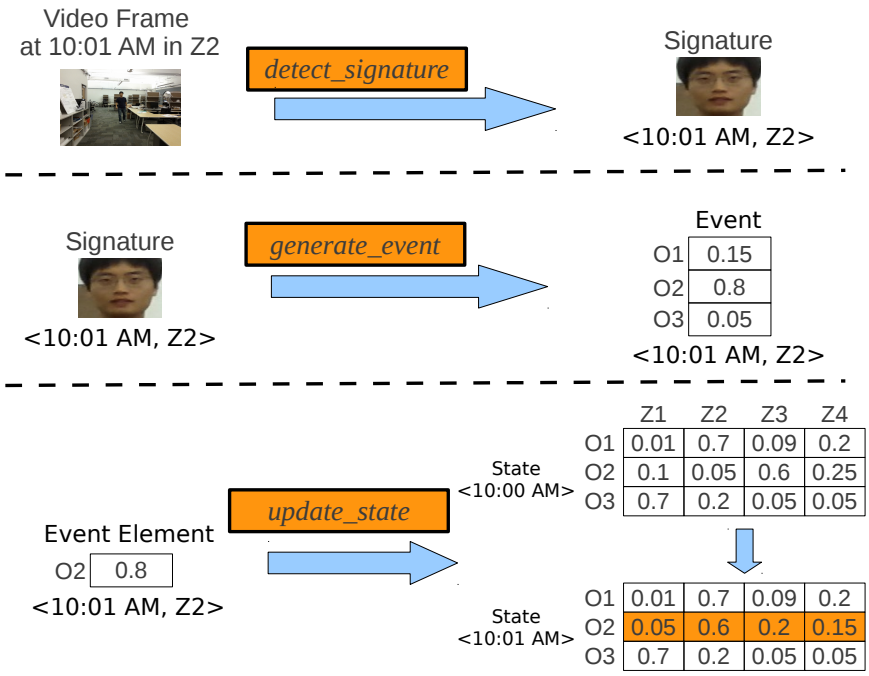


Figure 10: Roles of Spatio-temporal Analysis Handlers

with the current wall clock timestamp and zone ID of the camera by the runtime system. The runtime system also guarantees that the handler is invoked sequentially for images from a single camera stream. This allows a domain expert to use a stateful video analytics algorithm in the handler (such as adaptive algorithms that distinguish background and foreground [19, 56] to avoid detecting face-shaped background objects as faces).

Once a signature is captured, the *generate_event* handler is invoked. This handler generates a single event upon processing the detected signature. The runtime system automatically tags the generated event with the timestamp and zone ID derived from the input signature. Thus the programming abstraction automatically provides propagation of temporal causality in the spatio-temporal analysis application.

An event in this programming abstraction is an array of an application-specific data, where each element of the array is associated with a known occupant ID. For example, *generate_event* algorithm could be a face recognition algorithm that compares

the signature (a detected face) to known faces, generating an event that contains the application-specific similarity metric of the detected face to the known faces. Intuitively, a single event generation seems to be parallelizable since different comparisons between the detected signature and the known signatures are independent of one another. However, an event generation algorithm may have an arbitrary structure including sequential parts, which makes it tricky to automatically parallelize the handler execution.

For example, the Eigenface [69] algorithm uses principal component analysis (PCA) [75] to transform a face image to an eigenface before comparing different eigenfaces. In fact, PCA takes most of the execution time in generating an event, which makes it less attractive to parallelize just the comparison part of the algorithm. Other algorithms may want to normalize similarities between the detected signature and the known signatures [8], which involves a sequential process after the comparisons are completed. For these reasons, and to keep the programming effort of the domain expert simple, we do not attempt to parallelize a single event generation. In other words, *generate_event* handler is a sequential algorithm. However, the event generation for each detected signature can be executed in parallel. Therefore, we exploit parallelism at the level of multiple event generations, instead of different comparisons in a single event generation.

For each generated event, the application state should be updated to show the temporal evolution of occupants in different zones. Figure 10 shows how the *update_state* handler updates an application state from the current state to the next state. The application state is a table (two-dimensional array) of application-specific data indexed by occupants and zones. In this table, each row is called an *occupant state* since it gives the information of the whereabouts for a specific occupant; each column is called a *zone state* since it gives the information about the known occupants in a specific zone. As should be evident, each occupant state is independent

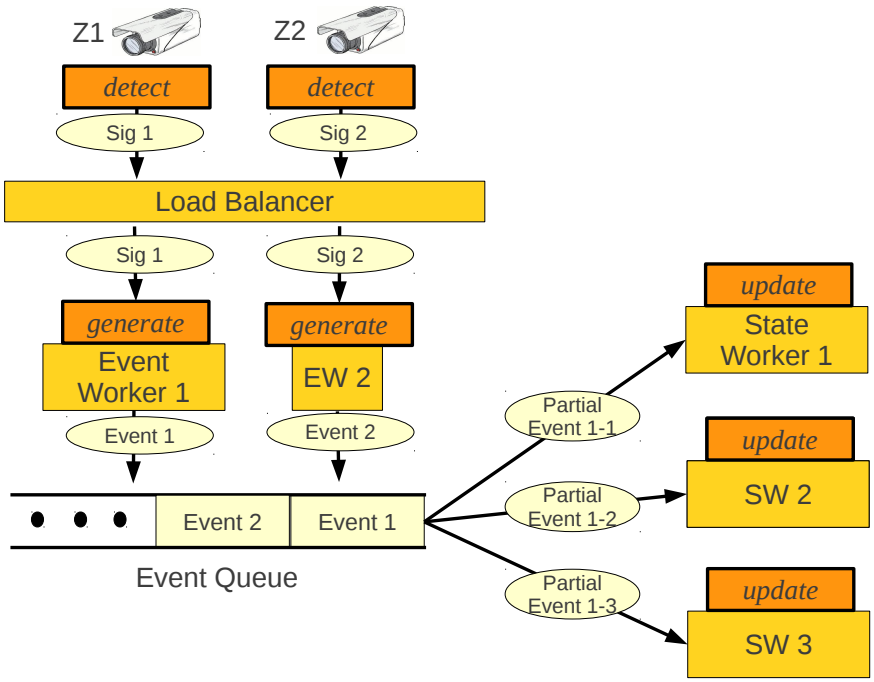


Figure 11: Execution of Spatio-temporal Analysis Handlers on Distributed Nodes

since movement of different occupants are independent. However, elements in a single occupant state are coupled. For example, a high probability in a specific zone would result in low probabilities in other zones. Based on this observation, the *update_state* handler is designed to update a single occupant state upon invocation, allowing our framework to exploit the inherent parallelism of state update. For a single event, a set of *update_state* handlers can be invoked in parallel, on different occupant states with different elements of the event (Figure 10). This makes it possible to distribute the workload of state update over distributed computing nodes.

4.2.2 Distributed Execution of Handlers

Once handlers are registered, they are invoked by the runtime system on each of the distributed nodes. Figure 11 shows the execution of the handlers on the distributed nodes. The *detect_signature* handler is invoked for each video frame at the distributed smart cameras. When a signature is returned by *detect_signature*, our system delivers it to any available worker node in the cloud, called an *event worker*. Different

signatures detected from a single smart camera can be delivered to different event workers in order to achieve load balancing across distributed event workers. At each event worker, *generate_event* handler is invoked with the input signature to generate an event. While events are independently generated at different event workers, the events are globally ordered based on their timestamps. Respecting the global temporal order, different elements of an event are delivered to specific distributed worker nodes called *state workers*. We will further discuss how our framework performs distributed state update on these state workers in the following section.

4.3 Scalable State Update

The workload of signature detection and event generation are massively parallel, since each signature and the associated event can both be independently computed on distributed nodes including smart cameras and cloud computing resources. This makes the two steps linearly scalable with the amount of distributed computing resources. However, state update becomes a bottleneck in a large-scale scenario due to the sequential update on a global state. This section discusses our solutions that solves the bottleneck by distributing computation cost and reducing communication cost of state udpate.

4.3.1 Distributed State Update

Unlike the previous two steps, state update requires sequential processing of events due to the inherent nature of maintaining a single global application state. Due to the probabilistic nature of the global application state, an event update potentially affects the probabilities of every occupant in all the zones. Therefore, to allow the temporal evolution of the global application state, each event has to be applied sequentially in temporal order to the current global state. In a large-scale situation awareness application, a centralized approach that maintains the global state at a single node will be overloaded due to the computation and communication overheads of state

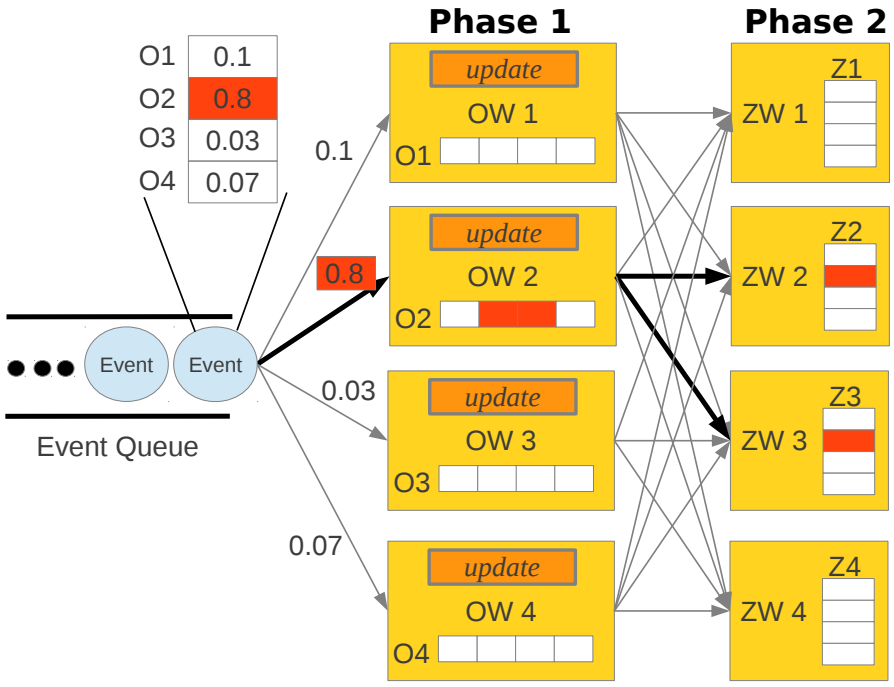


Figure 12: Distributed State Update for Spatio-temporal Analysis

update. If many events are generated at the same time, a single node cannot process all events in real-time and therefore the latency for situation awareness will increase.

To address the computation overhead of state update, we propose a distributed state update using partitioned application state across multiple state worker nodes where state update is performed on the partial application states at each node. Figure 12 shows our distributed state update using multiple occupant state workers (OW) and zone state workers (ZW). Each state worker maintains a set of occupant states and zone states to answer queries regarding specific occupants and zones. For example, occupant state worker OW1 maintains the occupant state of O1 to answer location queries on the occupant, while a zone state worker ZW1 maintains the zone state of Z1 to answer occupancy queries for the zone.

When a new event is generated, each probability for different occupants in the event are delivered to different occupant state workers commensurate with the occupant states that each worker is responsible for. Upon the arrival of each event

element, state update is performed on each occupant state at different occupant state workers (phase 1). Once the occupant states are updated, each occupant state worker transmits elements of occupant states to zone state workers (phase 2). As shown in the figure, the real work of computing the new probabilities for each occupant in every zone is carried out by the occupant state workers in phase 1. Phase 2 is a simple data copy of the computed probabilities in phase 1 and involves no new computation. Since no computation happens at zone state workers, phase 2 may seem to be optional. However, the zone state workers are crucial to handle occupancy queries efficiently because a user has to broadcast a zone-related occupancy query (e.g., who are in the zone X?) to all occupant state workers if there are no zone state workers.

4.3.2 Selective State Update

Although computation overhead is distributed over multiple nodes, communication overhead of state update is still a significant bottleneck. As Figure 12 indicates, each element of an event has to be transmitted to all occupant state workers, which increases the number of messages when more occupant state workers are used. Furthermore, each occupant state worker has to communicate to all zone state workers to update the zone states in phase 2. Assuming both *EventElement* and *StateElement* are double-precision floating-point values, the total communication cost in terms of bytes transferred linearly depends on the number of occupants and zones in a system:

$$Cost_{comm} = (N_{occupants} + N_{occupants} \times N_{zones}) \times sizeof(double) \quad (1)$$

For example, assuming thousand occupants and thousand zones, and 8 bytes for each double-precision floating-point representation, each event involves a communication payload of eight megabytes for state update. Assuming hundred signatures captured from distributed cameras every second, state update would incur a communication cost of eight hundred megabytes per second. Such a high communication cost

increases end-to-end latency of spatio-temporal analysis if the network infrastructure does not provide enough bandwidth among distributed worker nodes. Even worse, the required bandwidth for state update keeps changing over time, as it depends on the number of events generated in the system.

To solve the communication overhead, we propose a selective state update mechanism. In a realistic application scenario, the event generation algorithm (e.g., face recognition) may generate an event that has only a few significant probabilities. If an event generation algorithm is highly accurate, i.e., giving high probability for the ground truth identity and very small probability for other identities, a threshold can be applied to use only meaningful event elements for updating specific occupant states. Similarly, another threshold can be applied to occupant states to allow occupant state workers to transmit only significant changes in occupant states to zone state workers. Highlighted elements and arrows in Figure 12 show an example of selective state update. In the example, our system selects only one event element for O2 with significant probability, which is used for state update. After updating an occupant state of O2, only two significant changes for zone Z2 and Z3 are selected and transmitted to corresponding zone state workers. As shown in the figure, the communication cost of state update depends on selected occupants in each event and selected zones from each occupant state rather than the total number of occupants and zones in the system.

To allow such selective state update, our system provides two parameters to users: *occupant selectivity* and *zone selectivity*. Occupant selectivity allows a user to specify the number of occupants to be selected from each event, while zone selectivity specifies the number of zones to be selected from each occupant state. When an event is generated, our system finds occupants with top N probabilities pertaining to the occupant selectivity. Similarly, when an occupant state is updated, our system calculates the difference between the current state and previous state to selects zones with

top N changes. Our system supports automatic tuning of an occupant selectivity or a zone selectivity, which makes sure that the total communication cost is bounded by a user-provided threshold. To use the automatic tuning, a user specifies one selectivity (either occupant or zone selectivity) and the maximum communication cost. Using Equation 1, our system automatically infers the right value for an unspecified selectivity to make sure the total communication cost stays below the given maximum communication cost. While event rate changes over time, our system adaptively changes the unspecified selectivity to help system running in real-time in the presence of highly varying event rates from the real world.

4.4 System Evaluation

In this section, we evaluate our system through a set of experiments. In particular, we investigate the scalability of event generation and state update, and provide detailed cost breakdown for state update to show the reason of its bounded scalability. We also evaluate our selective update mechanism in terms of system-level performance and application-level accuracy.

4.4.1 Scalability of Event Generation and State Update

To evaluate scalability of spatio-temporal analysis, we performed a stress test on event generation and state update. Specifically, we increased event rate on a specific resource configuration until the latencies saturate; the system is overloaded from that point. We define the maximum event rate for the specific configuration as the event rate right before the latency saturation. We use the LBPH face recognition algorithm from OpenCV [9] in *generate_event* handler and the spatio-temporal update algorithm from Menon et al. [51] in *update_state* handler. For computing resources, we used *m1.medium* class nodes in Amazon EC2.

Figure 13 shows the maximum event rate of event generation and state update. Event generation scales well since events are generated on different workers, and

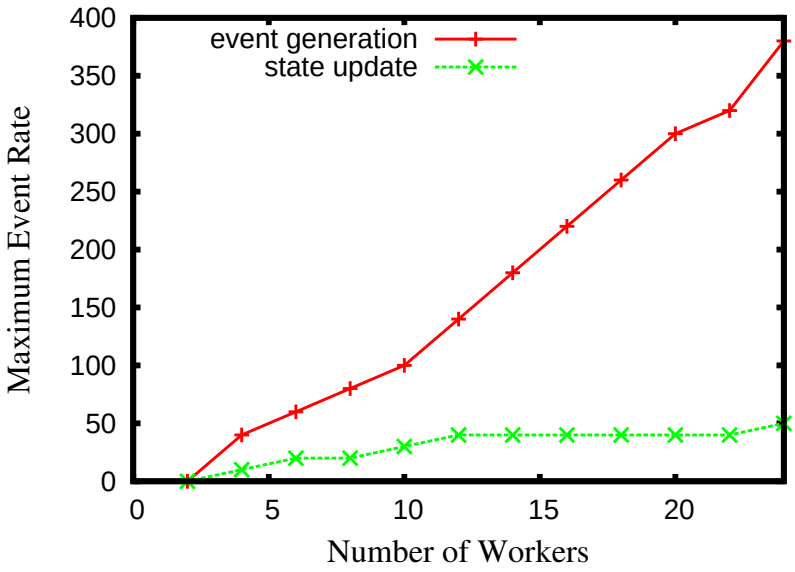


Figure 13: Maximum Event Rates with Different Number of Worker Nodes

putting more event workers will linearly increase the maximum event rate. However, state update without selective heuristics has a poor scalability since the maximum event rate does not increase with more workers.

To investigate the reason for the poor scalability of state update, we measured the detailed cost for state update. Figure 14 shows the total latency, computation latency and network latencies of state update for different number of state workers. The event rate (10 events per second) is carefully selected so that all configurations are not overloaded for the event rate. As the graph shows, computing latency decreases when more worker nodes are used. However, the network latency starts to increase from 16 worker nodes , which makes the total latency also increase from 20 worker nodes. This shows that the scalability of state update is bounded because the network latency starts to dominate from a certain number of state workers.

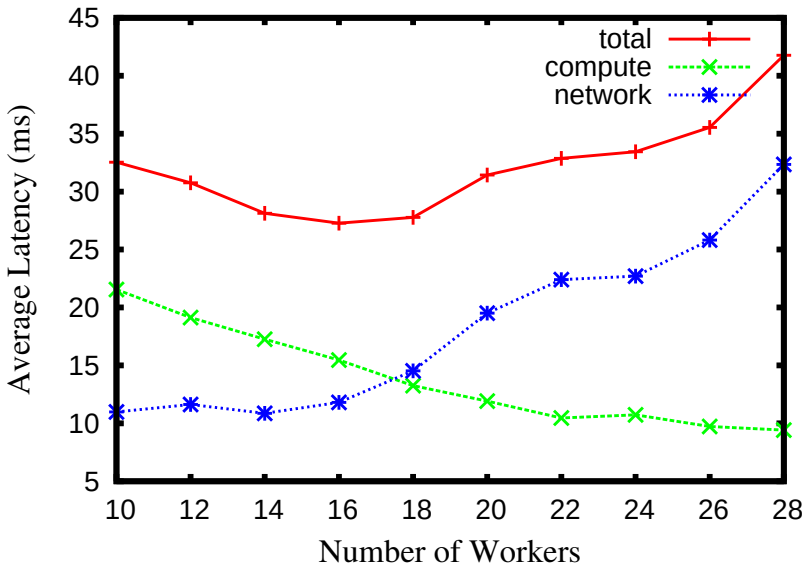


Figure 14: Average Computation and Communication Latency for State Update

4.4.2 Impact of Selective State Update

To measure the impact of selective state update, we performed the state update algorithm reported by Menon et al. [52] on eight distributed worker nodes in Amazon Elastic Compute Cloud (EC2) that are *m1.medium* class.

Impact of Selectivity on Latency. Figure 15 shows average latency of state update per event with different selectivity of occupants, while varying the scale of the system in terms of the number of occupants in the system. For example, *select-1* indicates selecting only a single occupant from each event while probabilities for all other occupants are ignored. Similarly, *select-all* indicates that probabilities for all occupants are used for state update. While varying the number of occupants, we fixed the number of zones to 1000. As shown in the figure, the naive state update selecting all event elements (select-all) scales poorly, as the system is overloaded when there are more than 500 occupants in the system. Until 500 occupants, latency for state update increases depends on the total number of occupants in the system. Other

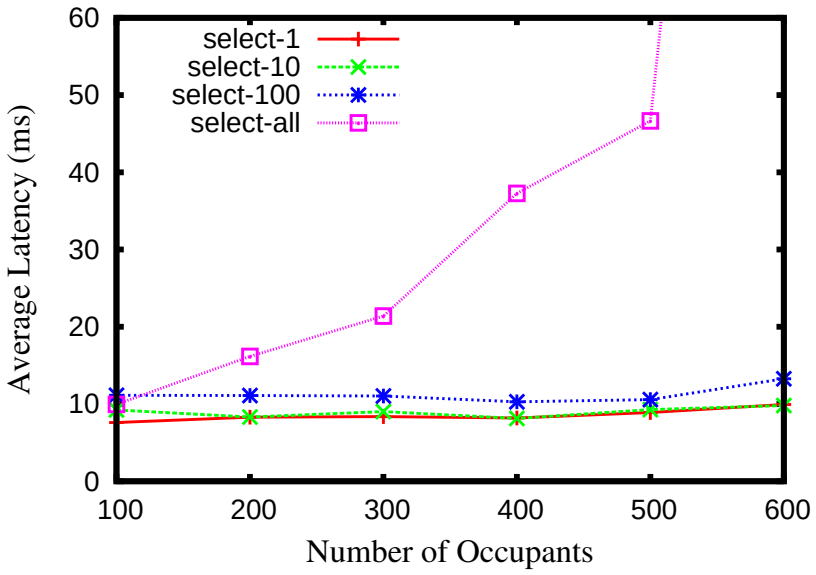


Figure 15: Average Latency for State Update with Occupant Selectivity

selective mechanisms show good scalability, where the average latency depends on the number of selected occupants rather than the total number of occupants in the system.

Similarly, Figure 16 shows the poor scalability of naive state update and improved scalability and latencies with selective zones for state update. In this case, we fixed the number of occupants to 425. With more than 1200 zones, the naive state update becomes overloaded and cannot handle incoming events in real-time. Other selective state update mechanisms scale well, while the latency for state update depends on the number of zones selected from occupant states. Because we used virtual machines in EC2, available bandwidth and communication latency between distributed state workers vary over time, which results in slightly nonlinear latencies in figures.

Impact of Selective Update on Spatio-temporal queries. In this experiment, we show the impact of selective state update on spatio-temporal queries using simulated events that are generated from simulation of moving occupants in camera

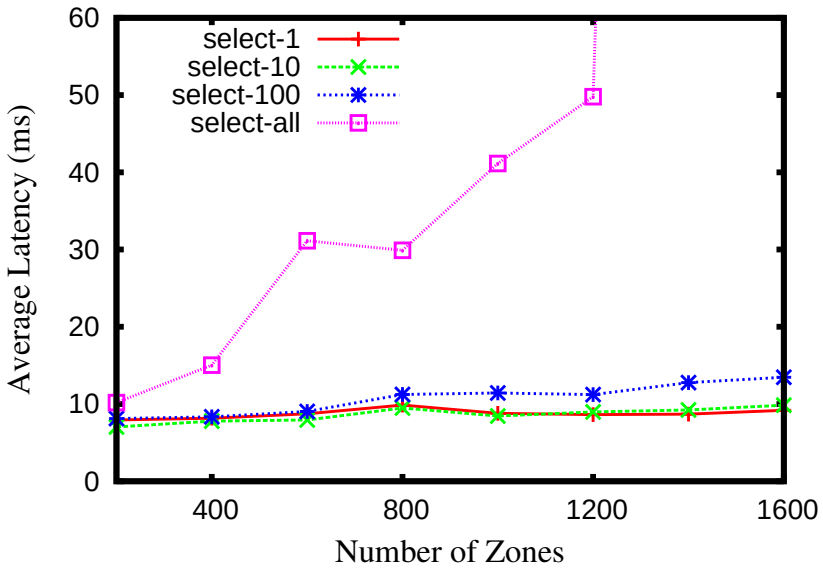


Figure 16: Average Latency for State Update with Zone Selectivity

network [1]. Since selective update ignores small probabilities in events and negligible changes in occupant states, resulting application state can differ from the application state computed by the naive state update. Although the naive state update does not always guarantee correct answers due to its inherently probabilistic nature, we use the naive state update as a baseline to compare with our approximated application state resulting from selective state update.

In this experiment, we used two different types of queries. First type of query, called *location query*, asks whereabouts of a particular occupant. For instance, an application may ask top three most likely places for an occupant in order to select potential video streams to track the occupant. Another type of query, called *occupancy query* asks the occupants who are likely (with higher than 0.5 probability) to be in a specific zone. Using the two types of queries, we compare an approximate

[1] We simulated randomly moving occupants in a camera network with a grid topology while events pertain higher probability for a ground truth occupant and lower random probabilities for others.

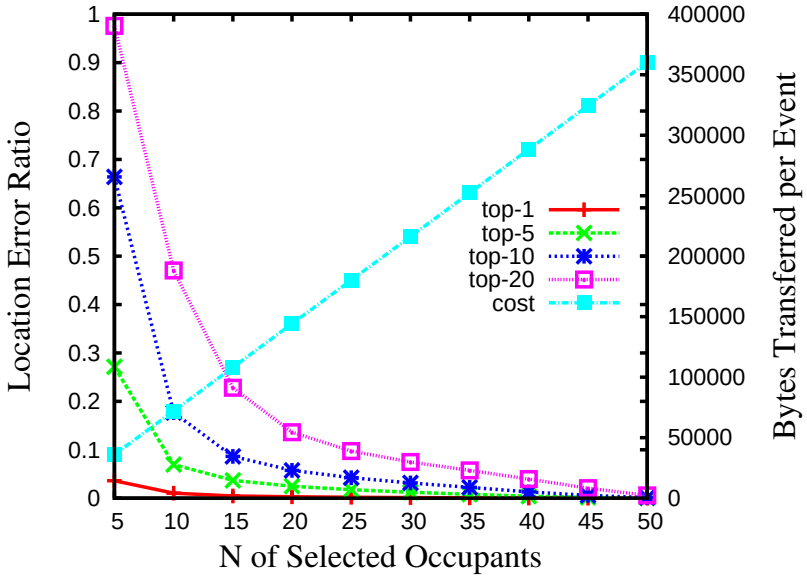


Figure 17: Accuracy of Query Results and Communication Cost with Occupant Selectivity

application state calculated by selective state update to an application state calculated by the naive state update. For each state update, we issue location queries and occupancy queries for all occupants and zones on the original application state and the approximate application state. If results for the same query differ between the original and approximate states, we count it as an error. Finally, we calculate the ratio of errors over all the query results.

Figure 17 shows the impact of selective update on the result of location queries. This experiment includes four different types of location queries asking different number of probable locations, which can be used for different application scenarios. For instance, *top-1* asks the most likely location for an occupant while *top-10* asks ten probable locations for an occupant to increase the chance to find the occupant. Over all types of location queries, the error ratio reduces when more occupants are selected for state update. However, the communication cost also increases due to the increased

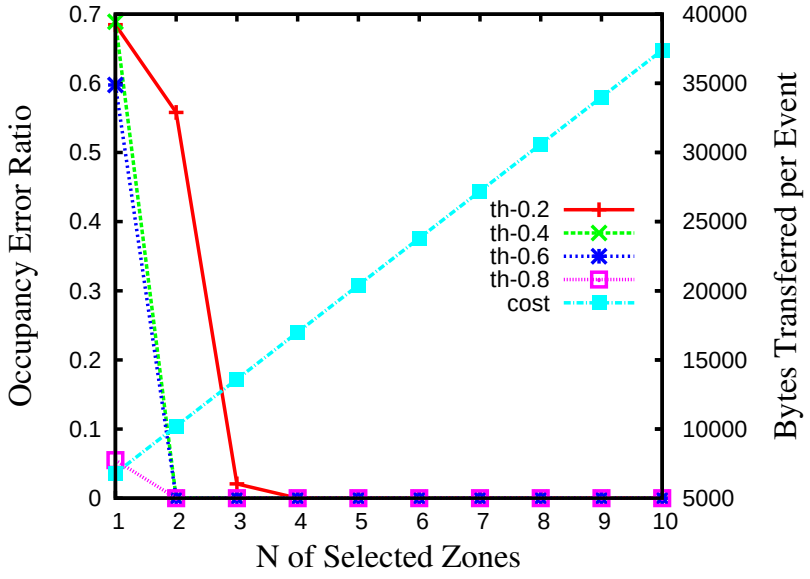


Figure 18: Accuracy of Query Results and Communication Cost with Zone Selectivity

number of messages transmitted from an event queue to the occupant state workers. When more probable locations are asked, the error ratio is higher since there is a higher chance of disparity between query results from an approximate state and the original state.

Figure 18 shows the error ratio for occupancy queries that ask probable occupants in a specific zone. For occupancy queries, we use a different threshold to answer probable occupancy, ranging from 0.2 to 0.8. Similar to the previous experiment, error ratio reduces when more number of zones are selected for state update while communication cost linearly increases.

Our experimental results shown in Figure 17 and Figure 18 indicate that using a small number of occupants and zones for selective state update can significantly reduce the communication cost without too much sacrificing the accuracy as measured by the error rates for the approximate state compared to the original state. For instance, if an application is interested in only the most probable location for each occupant,

using only ten occupants for the selective update is sufficient to achieve the same accuracy as compared to the naive state update.

4.5 Conclusion

Spatio-temporal analysis is one of the key inference techniques that convert raw video streams to knowledge of occupants' whereabouts, enabling a wide range of applications such as surveillance, transportation, assisted living, and the like. However, there are serious impediments to scaling such techniques to large-scale camera networks, as it involves processing a large number of events and sequential updates on a global state.

In this work, we tackle the technical challenges for developing a real-time spatio-temporal analysis application on a large-scale camera network. We present a novel distributed programming abstraction that minimize the burden on the domain experts by requiring them to provide the domain-specific handlers. Our system solves the performance problem of state update in large-scale scenarios by providing tuning parameters (i.e., occupant and zone selectivity) for reducing communication overhead between worker nodes. We have implemented and evaluated realistic applications using our system to show the scalability and efficacy of our programming abstraction.

CHAPTER V

OPPORTUNISTIC EVENT PROCESSING

This chapter presents an opportunistic event processing mechanism [33] that provides low-latency situational information for mobile situation awareness applications. Mobile situation awareness applications provide situational information to mobile users based on the events from various sensors that are widely deployed in the environment. For example, a connected vehicle application can notify drivers of live road conditions near them, warning of traffic, accidents, or obstructions on the road. These applications are naturally event-based, processing primitive events from sensors using application-specific algorithms to generate high-level events including situational information, and finally either reporting the situational information to a human or using it for automated decision making. In this context, events are spatio-temporal in nature – they occur at a particular place and time – and mobile users typically make continuous queries about their surroundings, i.e., situational information based on recent, nearby events.

As a specific example, consider a user who is driving from New York to Los Angeles. The user may have a vehicular application to automatically detect driving conditions along the route (e.g., traffic, accidents, road obstructions, or construction) and reroute the user around those problems. However, it is not practical to process the events along this whole cross-country route for the entire trip. Furthermore, an accident along the route near LA may not be relevant if it happens while the user is only just leaving NY. Therefore, the application should only process events in the vicinity of the user, according to some reasonable, application-defined range. Timely delivery of events is critical for this application, however, since there is no

point in notifying the user's vehicle of an accident or bad traffic after the user passes the last exit on the highway before the problem – it is too late for the user to do anything useful with that information. This is an example of an application where an approximate result is better than a late result.

Complex Event Processing (CEP) is a well-known paradigm to generate such situational information. To generate situational information, CEP applications use operator graphs consisting of multiple operators that perform online processing of events. Online processing of events allows asynchronous, low-latency generation of situational information since events are processed as they come. In a traditional, infrastructure-based CEP application, a single operator graph would take input from all the sensors in the infrastructure and perform continuous computation to generate live events.

However, for a mobile situation awareness application, it is not scalable to continuously perform live computation on all of the sensors everywhere. Furthermore, mobile users are typically only interested in events occurring within a certain area around them. Computing events outside that area results in wasted computation. The reduction in needed computation when only processing events in a range around the mobile user, vs. processing all events, is substantial [44]. A naive solution would be to start the operator graph anew in different locations as the mobile user moves to those locations. However, processing of some historical events is typically necessary before live event processing can begin. Often the user is interested in recent events, not only ones happening right now. It also may be the case that some historical context is needed to correctly process new events. Events must be processed in temporal order, so there is a processing delay to deal with the historical events before live event processing begins. Therefore, if the operator graph is started in a location only when a user moves to that location, the user will then be in a different location by the time the operator graph has finished processing the historical events. This

could potentially lead to a situation where the operator graph is constantly trying to "catch up" to the user, resulting in processing events in inappropriate locations and never actually getting to process any "live" events.

We propose to address this problem by processing the historical events for a certain location before the mobile user arrives at that location, so that live event processing begins at the moment the user arrives, if not before. Two existing technologies enable such a just-in-time live event processing: future location predictions for mobile users and processing time estimations for the CEP algorithms. Several location prediction algorithms already exist [76], and profiling techniques can be used to estimate processing time. Our system treats both of these as black boxes, allowing different location prediction and processing time estimation algorithms to be plugged in.

However, two important challenges still remain. First, if a mobile user is too fast compared to the processing time, historical event processing may take longer than the user takes to get to the new location. To mitigate this, we propose using parallel resources to enable pipeline processing of future locations in several time-steps look-ahead. Second, the location prediction algorithm may not provide a single, accurate location prediction. To mitigate the imperfection of location prediction algorithms, we propose taking several predictions for each time-step look-ahead and opportunistically process events in multiple predicted regions. When the user arrives at the future location, the processing result that is closest to the user's actual position will be returned to the user.

Our research contribution includes 1) a system architecture that enables spatio-temporal event processing for mobile situation awareness applications; 2) methods for (a) starting event processing at predicted future locations in advance of a mobile user's arrival, (b) pipelining multiple future prediction points to allow completion of event processing by the time the mobile user arrives, and (c) opportunistically processing events in multiple regions to provide accurate results to mobile users; 3)

Table 2: Query Parameters for Mobile Situation Awareness

Query Parameter	Description	Example
spatial interest	a mobile user's interested region, in terms of distance from the user's current location	500 meters from here
temporal interest	a mobile user's interested time duration, in terms of duration from current wallclock time	recent 5 minutes
operator graph	operator graph implementing situation awareness logic	Figure 19
location sensitivity	distance threshold to update the scope of situation awareness (i.e., Our system switches to a new operator-graph if the user moves more than this threshold from the previous location.)	100 meters

metrics for assessing the quality of results and timeliness of mobile situation awareness applications; and 4) an experimental evaluation of our system and methods.

5.1 Query Model for Mobile Situation Awareness

To support mobile situation awareness, our system collects various types of sensor data called events from widely deployed heterogeneous sensors such as smart phones, connected vehicles, and traffic monitoring cameras. Every event must have a set of required properties including type, location, and timestamp. Location and timestamp properties can be either a point or a range, regarding the type of event. The required properties are set by a producing sensor, specifying where and when a certain type of event is generated. Besides the required properties, each event may have optional application-specific properties for sensor data, which can be either structured (e.g., integer, string) or unstructured (e.g., audio or video).

Using events collected from various sensors, our system provides situational information to mobile users. Situational information is generated by executing an

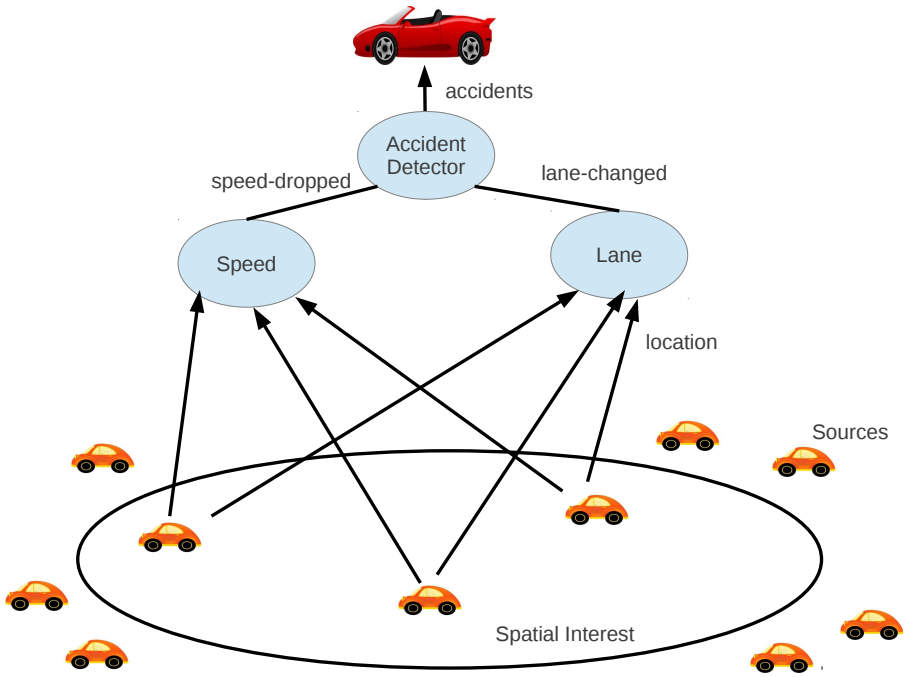


Figure 19: Example Operator Graph for Mobile Situation Awareness

application-specific operator graph on the collected events. An operator graph consists of multiple operators, where each represents a piece of computation. Each operator takes one or more input types of events and produces an output type of event. Connecting edges between operators define the logical flow of events. Take Figure 19 as an example of using an operator graph for mobile situation awareness. In the example, each car continuously reports location events, including its identifier and geographical position, to the operator graph. By consuming speed and location events, two leaf operators in the operator graph detect speed-patterns and lane switches of each car. The root operator is an accident detector, which incorporates the speed and lane information to detect an accident if many cars reduced their speed and avoided a specific lane at the same time. The final outcome of this operator graph, accidents, are the situational information that our system delivers to mobile users. Note that a real application may use heavy-weight operators (e.g., computer vision algorithms)

with unstructured sensor data (e.g., video frames).

To receive situational information, a mobile user registers a continuous query with a number of parameters listed in Table 2. An *operator graph* is the application-specific situation awareness logic that generates situational information from sensor events. The *spatial interest* and *temporal interest* specify a space and time range based on the user's current location and the current wallclock time for selecting input events to the operator graph. For example, a navigation system may want to display all recent accidents at nearby locations. Generating such recent situational information requires processing both live and historical input events, which require a mobile user to specify her temporal interest based on the duration from the current wallclock time. The user also specifies her spatial interest based on her current location to receive most relevant information.

A mobile user also sets a *location sensitivity* that indicates a distance threshold for updating the spatial scope of the situation awareness based on the user's current location. For example, a car navigation system has to show up-to-date information while a car is moving. It can set the location sensitivity as one hundred meters so it can receive updated situational information for a new region whenever the user moves more than one hundred meters.

5.1.1 Temporal Ordering

Note that it is crucial that our system injects events to operator graphs in a temporal order. If events are not temporally ordered, the application logic of each operator has to go through past events that are already processed, and perform its algorithm again to find certain patterns using updated sequence of events. Take for example an operator that detects a linear drop in speed over the last few seconds. If events are delivered in the right temporal order, e.g., $\{speed, t_1, l_1, 30\frac{km}{h}\}$, $\{speed, t_2, l_2, 20\frac{km}{h}\}$, $\{speed, t_3, l_3, 10\ \frac{km}{h}\}$, the operator can simply compare the last two events with every

new event arrival to detect such a pattern. If events are delivered out of order, e.g., $\{speed, t_3, l_3, 10\frac{km}{h}\}$, $\{speed, t_2, l_2, 20\frac{km}{h}\}$, $\{speed, t_1, l_1, 30\frac{km}{h}\}$, the operator either detects an increase in speed (wrong result) or has to sort the events and process events again when out-of-order delivery is detected (redundant computation). Our model assumes that events from the same sensor are delivered to our system in temporal order using a FIFO channel (e.g., TCP socket). With this assumption, we assign timestamps to input events based on their arrival time and provide them to operators based on the timestamps.

5.1.2 Operator Graph Switch

When a user moves to another region, our system needs to provide updated situational information for the region. As proven by Koldehofe et al. [44], deploying operator graphs on demand based on user mobility is more efficient than deploying a vast amount of operator graphs for all possible regions. Consider again the example that a consumer is interested in accidents that happened in the last hour in a five kilometers perimeter. Given the user interest, an operator graph is processing up-to-date events from the current region. Once the user moves to another region, we may inject those events from the new region to the existing operator graph instead of deploying a new operator graph. However, this approach breaks the temporal order of event processing since we have to process historical events from the new region while the existing operator graph is already processing live events from the previous region. To continuously provide situational information while a user is moving, our system creates a new operator graph for the user's updated spatial interest and terminates the previous operator graph. The new operator graph starts processing historical events for a new region and asynchronously delivers situational information to the user. After processing all historical events, the operator can provide live situational information by processing live events from the region.

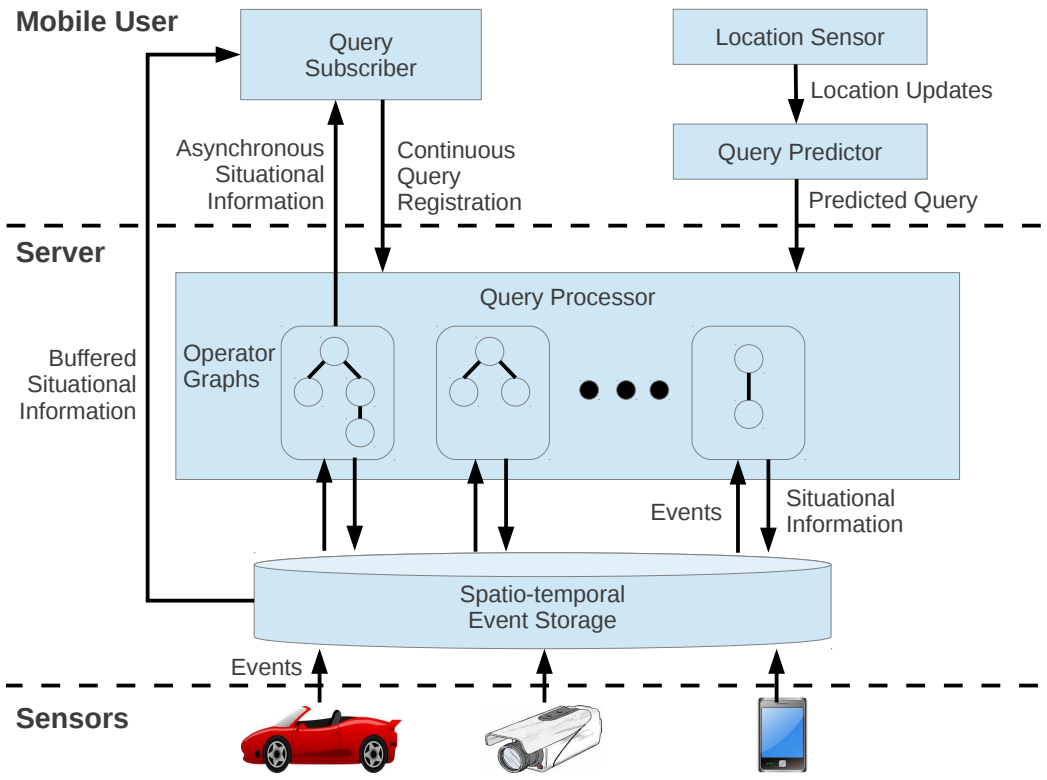


Figure 20: Logical Structure of System Architecture

5.2 System Architecture

Figure 20 shows the logical structure of the spatio-temporal event processing architecture. Each client has a *query subscriber* through which a mobile user registers a continuous query with a server. The registered query is handled by the server's *query processor* that creates an operator graph and executes it by injecting relevant spatio-temporal events matching the continuous query. While running, the operator graph generates high-level events for situational information, such as car accidents on highways, which are then asynchronously delivered to the current user through the query subscriber.

Our system includes a *spatio-temporal event storage* that stores primitive events from sensors as well as high-level events that are generated from operator graphs to serve future queries without redundant computation. All events in the spatio-temporal event storage are indexed by location, time, and type properties. An

administrator-defined time-to-live (TTL) value specifies a lifecycle of the events stored in the event storage. After an event is expired based on its TTL value and timestamp, our system removes the event from the spatio-temporal event storage. By removing old events, our system efficiently uses its storage space by keeping events that are more effective for situation awareness.

A *query predictor* is a key contribution in our system that allows opportunistic event-processing based on location predictions. The query predictor runs on each client device and sends requests for creating and running operator graphs on the future location of a mobile user. Detailed mechanisms of this component are discussed in the following sections.

5.3 Problem Formulation and Solution Overview

If a mobile user moves to a new location that is farther than its distance threshold (location sensitivity), our system creates a new operator graph with an updated region and starts processing historical events matching to the mobile user's temporal interest (e.g., recent 5 minutes). Although situational information is asynchronously generated and delivered to the user while processing the historical events, the mobile user can only receive live situational information after processing all historical events in temporal order. Because of the processing latency of historical events, switching to a new operator graph causes a delay before receiving live situational information. Since low-latency is a key requirement in mobile situation awareness, such a delay can be a significant problem. Another problem caused by the latency of processing historical events is the meaningfulness of situational information. By the time when recent situational information is delivered, a mobile user may already moved away from the previous location, which makes some situational information meaningless as they are outside of the current spatial interest of the mobile user.

To provide timely situational information, an operator graph must have processed

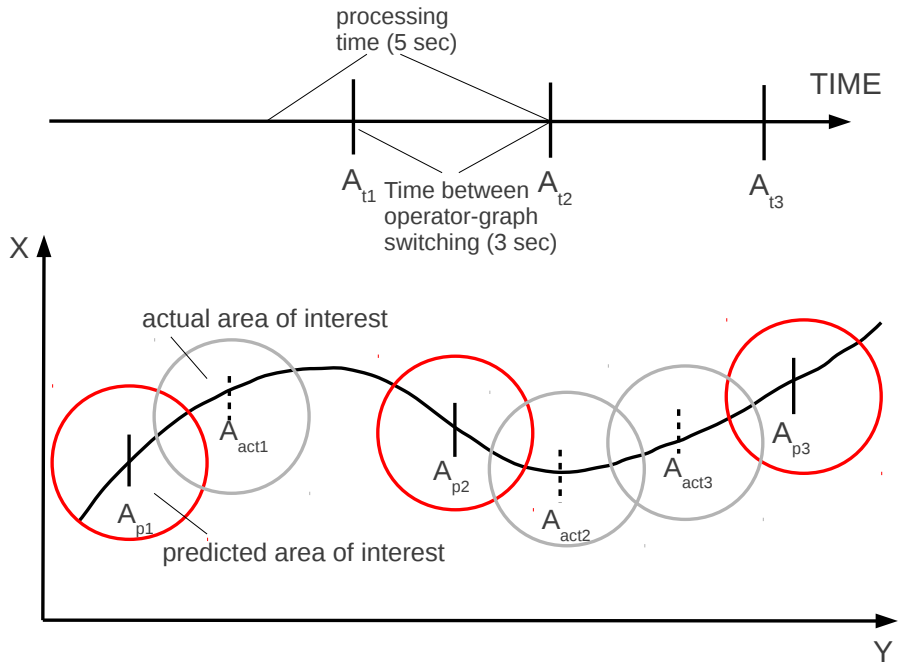


Figure 21: Mismatch between Predicted Query Regions and Actual User Interests

all historical events when a user switches to this operator graph. In the ideal case, there may be no historical events for the operator graph's region and therefore the processing latency for historical events is zero. However, if the region contains historical events matching to the temporal interest of the user, the operator graph should start earlier before the user switches to the operator graph, giving enough time to process all historical events. To start operator graphs earlier, our system performs opportunistic computing based on the location prediction of mobile users. Specifically, a location predictor provides a a set of future locations with probabilities. Using the predicted locations, our system creates operator graphs for each future location and starts running the operator graphs before a mobile user reaches one of the future locations.

The quality and timeliness of the resulting situational information highly depends on where and when an operator graph is deployed. Because location prediction algorithms provide uncertain future locations, it is highly probable that the actual spatial

interest of a mobile user does not match with the predicted region. In Figure 21, the predicted spatial region A_{p1} partially overlaps with the actual spatial interest A_{act1} since the prediction was inaccurate. In addition to the spatial uncertainty, the mobile user may pass through the predicted region before the operator graph finishes processing historical events. In Figure 21, the temporal axis shows that a consumer moves from A_{act1} to A_{act2} within three seconds while the processing historical events around A_{act2} takes five seconds. In this case, the resulting live situational information is less meaningful by the time it is delivered, although the location prediction was accurate.

To tackle the problem, we define metrics for the quality and timeliness of situational information. These metrics quantify the meaningfulness of query results when predicted query regions are different from the actual spatial interest of a mobile user (Section 5.4). Based on the metrics, we propose an event processing mechanism that predicts future locations of mobile users and initialize queries before user arrival to provide just-in-time situational information (Subsection 5.5.1). To avoid late delivery of situational information due to the high processing latency of historical events (e.g., in face of too many events in a query region with fast-moving clients), we extend the basic approach by pipelining future operator graphs for subsequent future locations (Subsection 5.5.2). We also provide a method that allows users to receive results with desired quality by over-provisioning operator graphs for multiple regions (Subsection 5.6).

5.4 Metrics

5.4.1 Quality of Query Result

We break the quality of results down to two metrics: *completeness* and *effectiveness*. Completeness indicates how many events in the actual spatial interest of a consumer are not included in event processing results, meaning false negatives. Effectiveness

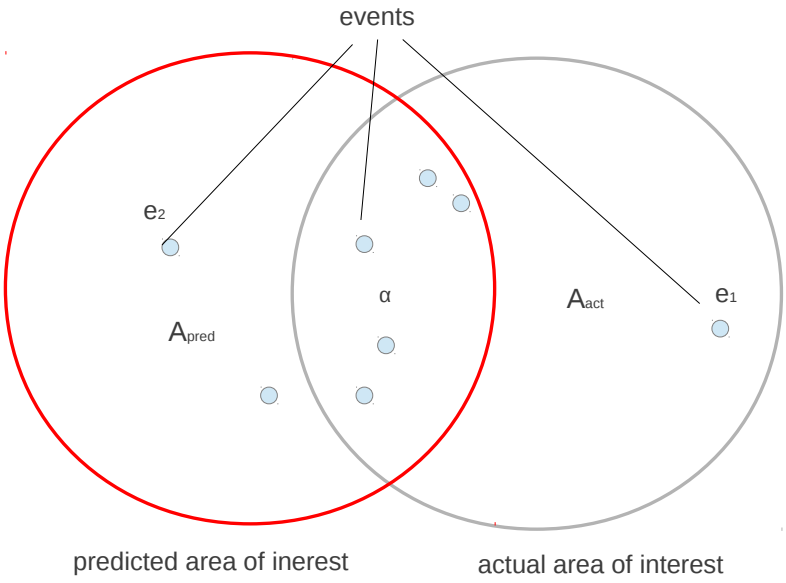


Figure 22: Quality of Results

indicates how many events are included in the processing but not covered by the actual spatial interest of a consumer, meaning false positives[1]

Since imprecise location predictions lead to overlaps between the actual spatial interest and predicted operator graph regions, not all events that lie in the spatial interest are included in the resulting information. For example, event e_1 in Figure 22 is not included in the area of the predicted operator graph although it is an interesting event for a mobile user.

An important observation, however, is that most of the relevant events in this example lie within the overlap. Completeness captures such partially included interesting events by representing a value close to one if most of the relevant events are in the overlap, while representing a value close to zero if most of the events are outside of the overlap. More formally, let V_{ov} be the number of events in the overlap between

[1]Note that these two metrics are similar to *precision* and *recall* in other domains. However, we defined our own metric based on the notion of overlapping events between a predicted query region and the actual spatial interest of a user.

the actual and predicted regions and V_i be the number of events in the actual spatial interest of the user:

$$completeness = \frac{V_{ov}}{V_i} \tag{2}$$

Effectiveness, on the other hand, takes into account those events that are not relevant to a mobile user (i.e., events outside of the actual region). For example, event e_2 in Figure 22 is not relevant to the mobile user but it is included in the result because of imprecise prediction. Effectiveness represents a value close to one if only few irrelevant events are included in the result, while representing a value close to zero if most of events in the result are not relevant to the user. More formally, let V_{ov} be the number of events in the overlap between the actual and predicted regions and V_p be the number of events in the predicted operator graph region:

$$effectiveness = \frac{V_{ov}}{V_p} \tag{3}$$

We can estimate the effectiveness and completeness for two circular regions A_{q_1}, A_{q_2} under the assumption of spatially evenly distributed events. Let α be the overlap of those areas, and er be the average event rate. Observe that the estimation is independent of the event rate:

$$E(completeness) = \frac{er * \alpha}{er * A_{q_1}} = \frac{\alpha}{A_{q_1}} \tag{4}$$

and

$$E(effectiveness) = \frac{er * \alpha}{er * A_{q_2}} = \frac{\alpha}{A_{q_2}} \tag{5}$$

5.4.2 Timeliness

The processing time for historical events depends on the complexity of the algorithm that is implemented in the operator graph, the available resources of the execution

platform, and the number of input events. In order to calculate approximate processing time for a given platform and the number of events, the operator graph can be profiled on different platforms with different number of events. After profiling, the processing time between those sample points can then be interpolated. Let $C_i()$ be the function that determines the interpolated time, T be the temporal interest, R be the spatial region of the operator graph, and $ev(R)$ be an average event rate for the spatial range. Based on these parameters, the anticipated processing time T_c can be computed as the following:

$$T_c = C_i(ev(R) * T) \tag{6}$$

Note that event processing results can be reused if an operator graph for the same spatial region is already deployed for another mobile user. Such reusing of results effectively reduces the processing time for historical events to zero.

5.5 Query Prediction

This section discusses an algorithm for predictive query processing. We first describe how the system predicts future locations and initializes the next operator graph each time the mobile user moves farther than the *location sensitivity* parameter (Figure 23). Thereafter, we extend the algorithm by initializing operator graphs for subsequent future locations to deal with late delivery of situational information (Figure 24).

5.5.1 Basic Query Prediction

Our system predicts future locations of a mobile user and deploys operator graphs based on predicted locations using an algorithm described in Figure 23. The algorithm deploys an initial operator graph (Q_{init}) when a mobile user initiates a query at the initial location. Henceforth, the system maintains the current operator graph ($Q_{current}$) that provides situational information to mobile users as well as a set of

```
1: L_previous ← L_init // previous location set to the initial location
2: Q_current ← Q_init // current operator graph set based on the initial location
3: Q_future ← ∅ // a set of operator graphs for predicted regions

4: upon locationUpdate(Location L_current)
5:  if L_current − L_previous > location_sensitivity then
6:      stopOperatorGraphs(Q_current)

        // select the next query based on user-defined policy
7:      Q_current ← selectNext(Q_future)
8:      discardOperatorGraps(Q_future − Q_current)
9:      deliverHistoricEvents(Q_current)
10:     initiateLiveNotification(Q_current)

        // initiate operator graphs for next future locations
11:     P ← getNextPredictedLocations(L_current)
12:     Q_future ← generateQueries(P)
13:     startOperatorGraphs(Q_future)
14:     L_previous ← L_current
15:  end if
16: end
```

Figure 23: Basic Query Prediction Algorithm for Mobile Situation Awareness

future operator graphs (Q_{future}) that process events for next expected locations of the user.

With every location update (e.g., GPS update), the query processor compares the current location ($L_{current}$) of the user to the previous location ($L_{previous}$) where the last operator graph switch happened. If these locations deviate more than *location_sensitivity*, the system stops the current operator graph and releases system resources associated with the operator graph (Line 5 – 6).

The system then selects one operator graph from the set of future operator graphs using a user-defined policy (Line 7). For instance, the system can select an operator graph based on the highest geospatial overlap, highest completeness, or highest effectiveness. Once selecting the best one, the system discards other operator graphs that are not selected and releases system resources that are associated with those operator graphs. (Line 8). The system then delivers all historical situations from the

```
1: P ← ∅ // set of predicted future locations
2: Q_current ← Q_init // current operator graph
3: Q_future[] ← ∅ // set of predicted operator graphs
4: currentStep ← 0

5: upon locationUpdate(Location L_current)
6:  if L_current − L_previous > location_sensitivity then
7:      stopOperatorGraphs(Q_current)
8:      switchToNewOperatorGraph(L_current,Q_future[currentStep])
9:      P ← L_current

        // deploy operator graphs on subsequent future locations up to eagerness steps
10:     for step ∈ [currentStep + 1, currentStep + eagerness] do
11:         P ← getNextPredictedLocations(P)

            // deploy new operator graphs if not exist or not accurate
12:         if notExists(Q_future[step]) ∨ locationDeviates(Q_future[step], P) then
13:             stopOperatorGraphs(Q_future[step])
14:             Q_future[step] ← generateQueries(P)
15:             startOperatoGraphs(Q_future[step])
16:         end if
17:     end for
18:     currentStep ← currentStep + 1
19:  end if
20: end
```

Figure 24: Pipelined Query Prediction Algorithm for Mobile Situation Awareness

spatio-temporal event storage that have been detected so far and afterward delivers live-situations detected by this operator graph (Line 9 – 10).

After selecting a new operator graph, the system deploys operator graphs for future locations (Line 11 – 14). The system retrieves a set of future locations from a location predictor and initializes operator graphs at each of those predicted locations. The number of future locations to use is a system parameter that is set by a system administrator. Section 5.6 explains a more sophisticated method for selecting future operator graphs with given system resources.

5.5.2 Pipelined Prediction

The algorithm described in the previous subsection initiates operator graphs for next predicted locations (i.e., possible locations where the next operator graph switch

may happen) to deliver just-in-time situational information. However, the algorithm causes poor quality of results if a user arrives at the next location before historical event processing for the next location is done. This may happen when a mobile user is too fast or historical event processing time is too long due to the large number of events around the next location.

To deal with the problem, our system pipelines prediction and initialization of operator graphs on subsequent future locations. Figure 24 presents the extended algorithm that performs such pipelining. Compared to the basic approach (Figure 23), the extended algorithm iteratively deploys multiple sets of operator graphs on subsequent future locations based on the *eagerness* parameter (Line 10 – 17). For every step in this iterative process, the location predictor is used to predict further-future locations based on the near-future locations. The intuition behind this approach is that one of those predicted locations will be selected to represent the actual location of a user. Note that uncertainty of location prediction would exponentially increase with the degree of pipelining (i.e., *eagerness*), which is a traded-off for timely delivery of results.

For each step of the pipelining, the system checks if operator graphs for the step are not deployed yet, or if already-deployed operator graphs deviate too much from up-to-date predictions (Line 12). If either case is true, the system initializes new operator graphs for the pipeline step (Line 14 – 15). When deploying new operator graphs because previously deployed operator graphs deviate too much, the algorithm stops those previously deployed operator graphs and releases system resources associated with them (Line 13).

5.6 Opportunistic Query Generation

Query prediction algorithms described in the previous section uses location prediction algorithms to retrieve future locations of mobile users. Since those predicted

locations are inherently uncertain, the system needs to select an appropriate set of operator graphs from predicted locations to provide good quality of results to mobile users. Specifically, the system should provide at least one operator graph that satisfies the user-defined quality requirements including completeness and effectiveness when switching to a new operator graph. To meet the quality of results, the system may deploy operator graphs opportunistically at multiple locations. However, because system resources are limited, the system cannot run operator graphs on all possible future locations. To deal with such constraints, the system generates future queries based on the resource limit represented in the maximum number of events that can be processed in parallel, as well as user-defined quality requirements.

To find such an appropriate set of operator graphs that meets both resource constraints and quality requirements, we first discretize the problem and then reduce it to a set coverage problem. For the discretization, we assume that only predicted locations are valid future locations of a mobile user, which is true if the number of predicted locations is infinite. The universe U for the set coverage problem is therefore the set of events covered by interest areas of operator graphs at all predicted locations. Given those events covered by each operator graph as a set of subsets G, the problem is then to select the minimum number of subsets $G' \subset G$, where all events in U are covered. However, we have to consider two additional constraints to the classical set coverage problem: 1) all areas of not selected subsets $N \subset G$ must overlap with areas of selected subsets $G' \subset G$, while completeness and effectiveness for all sets in N are expected to be ensured, and 2) the overall expected resource usage S_{tot} stays below the resource limit S_{max}.

Figure 25 shows the algorithm[2] that finds the appropriate set of operator graphs with given resource and quality constraints. The algorithm takes a set of predicted locations (P) as an input, then estimates the cost of each operator graph at each

[2]We adapted the greedy Johnson's Algorithm [39] to solve the set coverage problem.

```
1: function generateQueries( Locations P )
2:   Q ← ∅
3:   S_tot ← 0
4:   S[] ← calculateInitialCosts(P) // S[p] indicates the cost of an operator graph at p
5:   while P ≠ ∅ do
6:       p ← selectQueryLocation(P) // select a location based on user-defined policy
7:       P ← P − p
8:       // select if the location is not covered by other operator graphs
9:       if notCovered(OperatorGraph(p, Q)) then
10:          // add an operator graph only if resource limit is not reached yet
11:          if S_tot + S[p] ≤ S_max then
12:              Q ← Q ⋃ {OperatorGraph(p)}
13:              S_tot = S_tot + S[p]
14:          end if
15:      end if
16:  end while
17:  return Q
18: end
```

Figure 25: Opportunistic Query Generation Algorithm

predicted location (Line 4). Once the costs are estimated, the algorithm iteratively adds a new operator graph for each predicted location $p \in P$ into the set of future queries Q (Line 5 – 16). In each iteration, the algorithm uses a user-defined policy to select a location p from the set of predicted locations P (Line 6). Choosing p with the highest probability helps deploying operator graphs on highly probable locations, increasing overlaps between the actual user location and predicted operator regions. Choosing the operator graph with the lowest expected cost $S[p]$, on the other hand, helps deploying more operator graphs than the first approach, but possibly at less likely locations. With selected p, the algorithm checks whether p is already covered by other operators or not (Line 9). A location p is covered if any operator graph in Q already satisfies quality requirements when a user arrives at p in the future. If p is not covered and adding an operator graph at p does not exceeds the resource limit S_{max}, then the algorithm adds the new operator graph at p into Q, the set of future queries. The algorithm stops after checking all predicted locations (Line 5).

5.7 Evaluation

In this section, we evaluate our system by measuring the timeliness and quality of results based on realistic spatio-temporal events and user mobility. Timeliness is a delay between switching to a new operator graph and receiving live situational information after processing historical events. Our system predicts future locations and starts running operator graphs on the locations before user arrival, therefore provides near-zero latency for receiving live situation updates when switching to the new operator graphs. Quality of results are measured in terms of completeness and effectiveness as defined in Section 5.4. In our approach, the quality of results can degrade if location predictions are inaccurate. To compensate such prediction errors, we opportunistically deploy multiple operator graphs for future locations, increasing chances to find a better operator graph with higher quality of results. As provided in the subsequent sections, our system outperforms on-demand query processing that starts operator graph after user arrival both in terms of timeliness and quality of results.

5.7.1 Experimental Setup

To evaluate our system, we conducted simulations using SUMO [4], a well-known traffic simulator that generates realistic mobility patterns of vehicles on a real road network. Our simulations monitored the traffic in the downtown area (3.791 km x 2.872 km) of Atlanta for 20 minutes using the road network obtained from Open Street Map [27]. We originally simulated 1000 vehicles for each simulation but the random trip generator of SUMO automatically pruned out some invalid routes after generating trips, which resulted in 884 vehicles per simulation on average. For each simulation, we observed 282,951 events on average, meaning each vehicle reported 320 events on average.

During the simulation, we recorded each vehicle's geographical location at every

second, which is used in two different ways in subsequent experiments. In one case, we treated the location reports as anonymous spatio-temporal events that are used by operator graphs to generate situational information. In another case, we used trajectories of individual vehicles to simulate mobile users receiving situational information through continuous queries.

When measuring the quality of results and timeliness, we use four different event processing mechanisms, namely *zero*, *eager-oracle*, *eager*, and *lazy*. *Zero* represents an ideal case for both opportunistic event processing and on-demand event processing, which assumes zero computing cost for event processing. In this case, *location sensitivity*, a user-given parameter for the operator graph switch, is ignored and an operator graph is created upon every fine-grained location update. At each location, the created operator graphs provide up-to-date situational information immediately since the cost to process historical events is zero, resulting in perfect quality of results at any given time and location.

Eager-oracle is an ideal case for the opportunistic event processing that assumes an oracle predictor, which knows the exact future locations of all mobile users. Since our system knows the exact future locations, it can run operator graphs on the exact locations before user arrival. However, unlike the *zero* case, operator graphs are created at coarse-grained locations defined by *location sensitivity* since the computing cost is not zero. While a mobile user is keep moving, the user's spatial interest may be slightly different from the current operator graph region at each moment, resulting in degrades in quality of results. In the following experiments, *zero* provides perfect completeness and effectiveness while *eager-oracle* provides an upper bound quality of results for the opportunistic event processing mechanism.

Eager represents the opportunistic event processing with a realistic location predictor, called *dead-reckoning*. Dead-reckoning is a process of predicting future locations based on the current location and velocity of a moving object. We used simple

linear dead-reckoning that predicts future locations based on ten location histories. To compensate for decreased completeness due to the erroneous nature of location prediction, we create four opportunistic operator graphs for each prediction.

The last case, *lazy*, represents the on-demand event processing that creates an operator graph and starts processing historical events upon user arrival. Because of processing latency for historical events, a user cannot immediately receive up-to-date situational information, which results in degradation of both timeliness and quality of results.

5.7.2 Quality of Results Comparison

This section compares different event processing mechanisms by measuring the quality of results. We measured two metrics, completeness and effectiveness, at each fine-grained location of individual vehicles based on the overlapping events between the mobile user's actual spatial interest and the current operator graph's region. Although both metrics are measured, we only present completeness since they show the same trend. We also vary user-given parameters as well as processing latency for historical events in order to investigate each parameter's impact on the quality of result. For each experiment, the following default parameters are used, except one parameter that is selected as a control variable.

$$
\begin{aligned}
\textit{Location Sensitivity} &= 100 \text{ meters} \\
\textit{Spatial Range} &= 500 \text{ meters} \\
\textit{Temporal Range} &= 60 \text{ seconds} \\
\textit{Processing Latency} &= 4 \text{ seconds}
\end{aligned}
$$

Figure 26 shows completeness for different event processing mechanisms while varying location sensitivity. In this experiment, smaller location sensitivity means more fine-grained, and more frequent operator graph switches while a user is moving. As shown in the figure, both *eager* and *eager-oracle* provide better quality of results

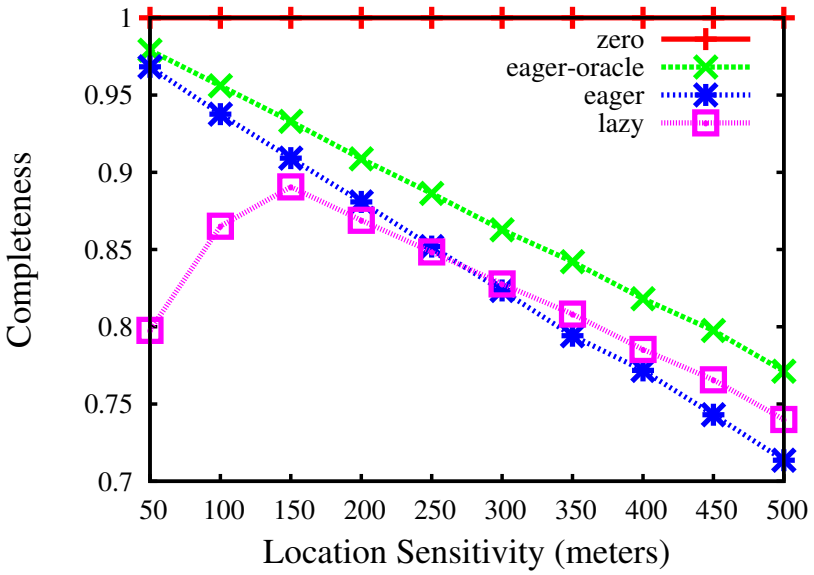


Figure 26: Quality of Result with Different Location Sensitivity

as location sensitivity decreases because the distance between two successive operator graphs depends on the location sensitivity, and the quality of service degrades as operator graphs are spread farther apart. *Eager* shows a steeper decrease in quality of results since the prediction error also increases when predicting far-away future locations. In contrast to opportunistic event processing mechanisms, on-demand event processing, or *lazy*, shows that it achieves peak completeness at 150 meters of location sensitivity. At small location sensitivities, we observed that an operator graph cannot catch up to the fast-moving vehicles since the vehicles moves away from the operator graph's region before the operator graph finishes processing historical events. On-demand event processing cannot handle such fast-moving vehicles with a small location sensitivity, thus showing the necessity of opportunistic event processing. At large location sensitivities, *lazy* shows a similar trend to *eager-oracle* because the quality of results decreases between two subsequent operator graphs while there is no uncertainty of future location involved.

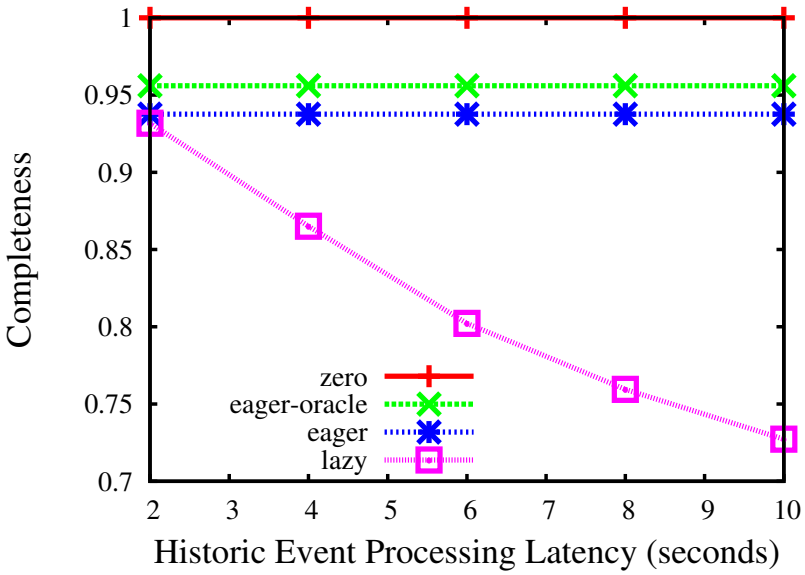


Figure 27: Quality of Result with Different Processing Latency

Figure 27 presents completeness while varying historical event processing latency. When a user switches to a new operator graph, the operator graph should process recent historical events to provide up-to-date situational information. In realistic scenarios, processing latency for historical events depends on the number of events and the complexity of operators. However, we used processing latency as a control variable in this experiment to show the impact of processing latency on the quality of results. As shown in Figure 27, *eager* and *eager-oracle* are not affected by historical event processing latency because they process events in advance through the opportunistic event processing mechanism. The difference in quality of results between the two is due to the imperfect location predictions in *eager*. However, *lazy* shows decreasing completeness when historical event processing latency increases. The decreased completeness is caused by user mobility, since a requesting mobile user can be far away from the requested location by the time a new operator graph is ready to begin "live" processing.

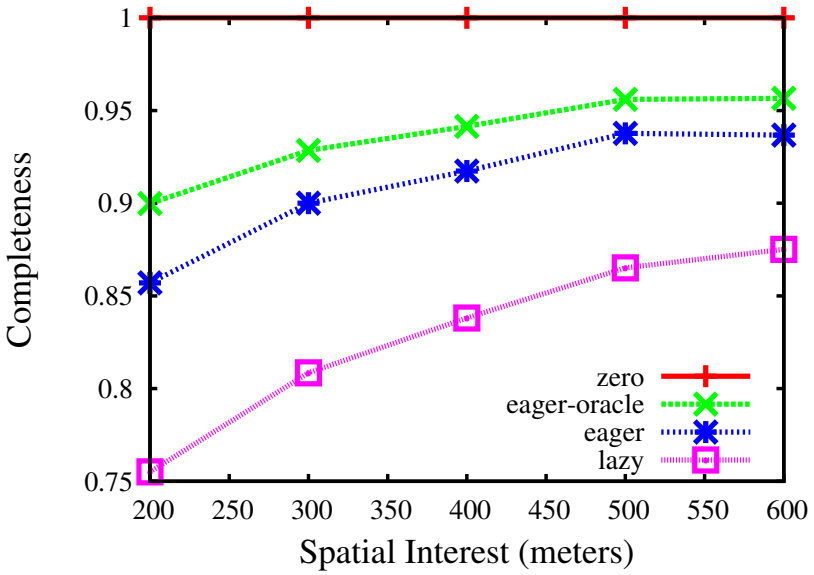


Figure 28: Quality of Result with Different Spatial Interest

Figure 28 shows changes in the completeness when the range of a mobile user's spatial interest changes. In this experiment, all event processing mechanisms show the same trend that wider spatial interest yields a better quality of results. Although quality of service decreases between two subsequent operator graphs because of missing events and unnecessary events, the number of events that are missed or unnecessarily included can be negligible at large spatial interest.

5.7.3 Timeliness Comparison

This section compares the timeliness of situational information between opportunistic event processing and on-demand event processing. We compare timeliness by measuring the delay between a mobile user's arrival at a new location and receiving live situational information about that location. In on-demand event processing, the delay is exactly the same as the historical event processing latency since live situational information is only available after processing all the historical events. In the opportunistic event processing mechanism, however, the new operator graph might already

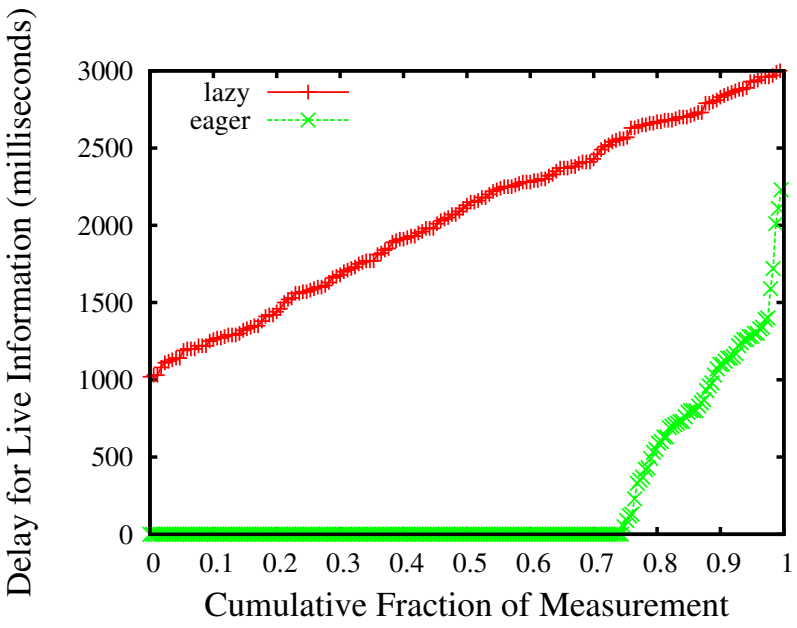


Figure 29: Timeliness of Opportunistic and On-demand Event Processing

be created and have processed all the historical events. In this case, the delay for live situational information is zero since the mobile user can immediately receive the information.

In this experiment, we used a dummy operator graph that takes a uniform random latency from one to three seconds to process historical events, while both the server and client modules are running on the same local machine. In a realistic scenario, timeliness will be also affected by network latency between a mobile user and a server that is running an operator graph. For opportunistic event processing, we use the *dead-reckoning* predictor to predict future locations while four operator graphs are opportunistically created at each prediction.

Figure 29 shows the cumulative distribution function (CDF) for the timeliness of both on-demand and opportunistic event processing mechanisms. On-demand event processing, labeled as *lazy*, creates an operator graph after user arrival and therefore it suffers from poor timeliness caused by the historical event processing latency. The

delay for receiving live situational information is uniformly distributed between one to three seconds, following the distribution of event processing latency. However, our approach of opportunistic event processing (labeled as *eager*) provides zero latency in more than 70% of time because the operator graphs have already been created and the historical events processed when the user arrives at the future location. In few cases, our system has to create operator graphs on demand because none of the predicted queries can satisfy quality requirements, which causes the same amount of delay with the on-demand event processing. Another cause of delay in opportunistic event processing is that a vehicle moved too fast and therefore the operator graph did not process all historical events yet.

5.8 Conclusion

The abundance of sensors in our environment enables mobile situation awareness applications that provide live situational information to mobile users. To provide up-to-date information, those applications need to update the spatio-temporal range of event processing as mobile users keep moving. However, processing historical events for each update of event processing region causes a delay to delivering live situational information.

In this work, we have proposed a system and methods that provide timely, yet highly relevant information to mobile users by avoiding historic event processing delay. Our contribution includes: 1) the system architecture that supports mobile situation awareness applications, 2) methods for eager computation of historical events, including a pipelining method to look ahead several steps into the future and an opportunistic computing method to compensate the inaccurate location prediction, and 3) an evaluation showing that our methods can achieve near-zero latency for mobile situation awareness while meeting the quality of result requirements.

CHAPTER VI

MOBILE FOG

The ubiquitous deployment of mobile and sensor devices is creating a new environment, namely the *Internet of Things* (IoT), that enables a wide range of situation awareness applications. For example, a participatory surveillance application can analyze user-provided live video streams from smart phones to track a suspect in real time. These applications have highly dynamic workloads over space and time since their workloads depend on the situation of physical environment. For instance, the number of data streams from a certain area changes over time due to user mobility, while the computational overhead for analyzing individual streams may also change depending on how many objects are detected in each stream. Situation awareness on IoT environment, therefore, requires handling highly dynamic workloads over space and time.

One possible solution to handle such dynamic workloads is using clouds as computing infrastructures. Clouds provide on-demand computing resources with a utility computing model, allowing applications to grow elastically over time. However, existing datacenter-based clouds are designed for traditional web applications, making them ill-suited for geospatially distributed, latency-sensitive applications. In particular, cloud-based approach requires data streams from widely deployed sources to flow through the Internet, imposing burdens on the network infrastructure. More importantly, applications suffer from high communication latency and jitters because of the network distance between edge devices and cloud computing resources.

To overcome limitations of existing datacenter-based clouds, Cisco recently proposed a new computing paradigm called *fog computing* [6]. The essential idea of fog

computing is to provide highly available computing resources throughout the network infrastructure, from the edge to the core of the Internet. In contrast to the cloud, these geospatially distributed, hierarchical computing resources allow applications to perform low-latency stream processing near stream sources. Applications can also use those resources to perform efficient in-network processing for aggregation and query handling.

While the fog computing paradigm presents a potentially useful computing infrastructure for situation awareness applications, managing computing resources in the fog is more complicated than in traditional computing infrastructures (e.g., clusters and data centers). Specifically, edge devices serving both sensing and computing resources are highly dynamic because of user mobility. Fog computing resources are also hierarchical, since they are deployed at different levels of network hierarchy (e.g., computing resources at access networks and those attached to core routers). They are geospatially distributed and highly heterogeneous as computing resources in different areas could be managed by different entities.

To support situation awareness applications on such a complex computing environment, we propose a fog-based execution environment for situation awareness applications, called *Mobile Fog* [32]. Mobile Fog supports three main functionalities as the following. First of all, it automatically discovers fog computing resources at different levels of network hierarchy and deploys application components onto the fog computing resources commensurate with the latency requirements of each component in the application. Secondly, it provides a hierarchical communication API that allows different components of an application to communicate each other. Lastly, it supports both latency- and workload-driven resource adaptation over space (geographic) and time to deal with the dynamism in situation awareness application.

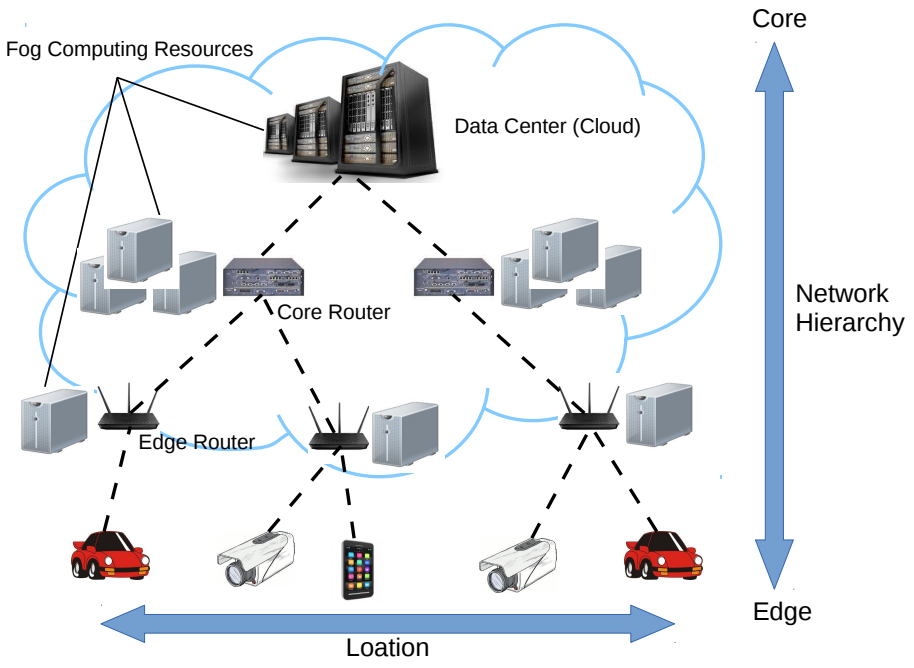


Figure 30: A Fog Computing Infrastructure with Computing Resources in the Middle

6.1 System Assumptions

Mobile Fog design is predicated on the availability of fog computing resources that are deployed throughout the network infrastructure. Figure 30 shows a fog computing infrastructure that includes computing resources attached to edge / core routers as well as computing resources in a cloud. In the figure, dashed lines indicate physical network paths between sensing devices and the cloud, where a single dashed line can be multiple network hops. The intuitive assumption for these fog computing resources is that both network latency and computational capacity increase as we go deeper in the network hierarchy from sensors to the cloud. We further assume that the fog computing infrastructure provides Infrastructure-as-a-Service (IaaS, e.g., Amazon EC2) interface that allows Mobile Fog to deploy arbitrary computation on fog computing resources. To make intelligent decisions for application deployment and resource adaptation, Mobile Fog needs to access fog computing resources in specific area and at certain level of the network hierarchy.

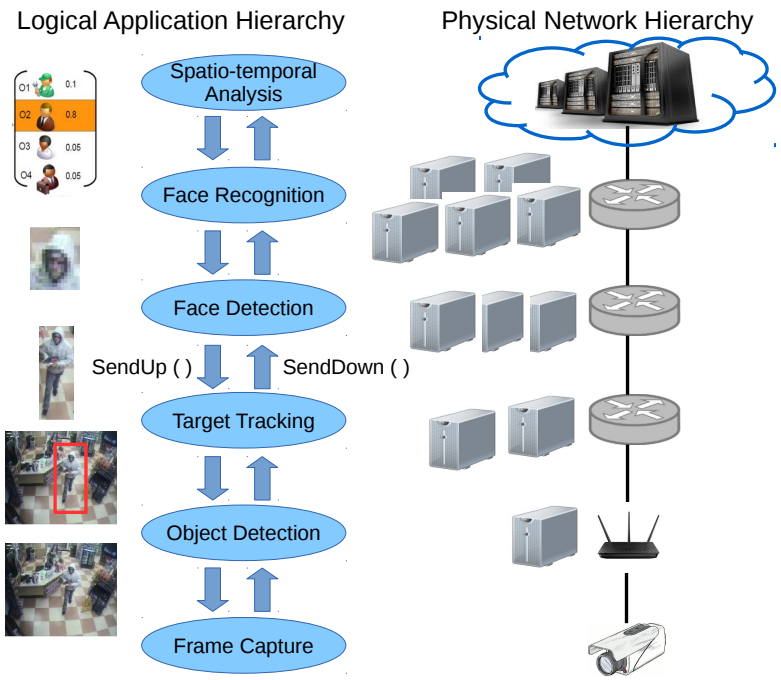


Figure 31: Application Mapped onto Physical Network Hierarchy

6.2 Application Requirements

Many situation awareness applications have multiple logical components that generate high-level actionable knowledge from unstructured raw streams (e.g., video streams). Figure 31 shows an example of such processing components in a surveillance application. As shown in the figure, the application performs five steps of processing on live streams to track multiple targets over different areas. The application 1) detects new objects in live video streams, 2) tracks targets in each video stream to extract individual target images, 3) finds faces from target images, 4) compares face images to the known set of faces to generate probabilistic events, and 5) updates an application state using those events to handle spatio-temporal queries.

In a situation awareness application, each logical component potentially involves dynamic workload depending on real-world situations. For instance, a target tracking component in Figure 31 has workload that depends on the number of targets in a live video stream. In general, such workload dynamism increases while going deeper

in processing pipeline because higher-level components analyze events from multiple streams over a wide area while lower-level components analyze individual streams. For instance, the spatio-temporal analysis component in Figure 31 involves event aggregation over a wide area while target tracking component tracks multiple targets in a single video stream. To deal with such dynamic workloads, we need to run application components on distributed computing resources in a scalable manner, especially for the higher-level analysis components.

In addition to handling dynamic workloads, low communication latencies between application components is critical to provide the good quality of service (QoS) for an application. For instance, object detection and target tracking components in Figure 31 need to share real-time target positions to avoid redundant detection of the same target. A target tracking component may also send command messages to a frame capturing component running on a PTZ camera to change the camera angle for a better view. In some cases, components need to stream a large amount of data (e.g., video frames) to other components, which requires low communication latency for high TCP throughput [55]. To support low-latency communication for application components that interact each other, it is essential to place those components onto nearby computing resources in terms of network distance.

To support low communication latency and dynamic workload handling, Mobile Fog deploys those components on fog computing resources at different levels of network hierarchy. Figure 31 shows an application components deployed on fog computing resources from sensing devices to cloud resources. In contrast to the cloud, putting application components in the network infrastructure allows low-latency stream processing near the edge, while highly dynamic, wide-area aggregation can be performed on elastic computing resources at the core of the network.

6.3 Resource Discovery and Application Deployment

To allow an application to execute on resources at different levels of network hierarchy, Mobile Fog automatically discovers computing resources and deploys application components on the resources. The intuition behind Mobile Fog's resource discovery and deployment mechanism is that both sources of data streams and consumers of actionable knowledge are 1) connected to the edge of the Internet, 2) geospatially distributed, 3) mobile, 4) ephemeral, and 5) generating dynamic workload depending on physical environments. In the following subsections, we discuss details of Mobile Fog's resource discovery and application deployment mechanisms that support applications to run on the fog computing infrastructure.

6.3.1 Dynamic Resource Discovery Protocol

To support highly dynamic stream sources and computing resources, Mobile Fog uses simple and scalable resource discovery protocol without any global knowledge. Upon a request for finding an upper-level computing resource (i.e., at initial *send_up()* or during migration either for load balancing or due to application mobility) Mobile Fog contacts a regional name server to receive a list of connection endpoints for computing resources in a certain area at a certain level of network hierarchy. Such regional name servers can be easily replicated for scalability since they only maintain connection endpoints that are updated infrequently.

Once a set of connection endpoints are received, a child node sends "ping" messages to a set of upper-tier computing resources to find out their availability. If a computing resource is available to accommodate the new child node, it sends a "pong" message back to the child node with its workload level. Using the workload level, a child node selects an appropriate parent node that is not overloaded. Mobile Fog currently uses the average waiting time (i.e., queuing latency) of recent stream data as the estimation of workload level. To ensure QoS requirements , Mobile

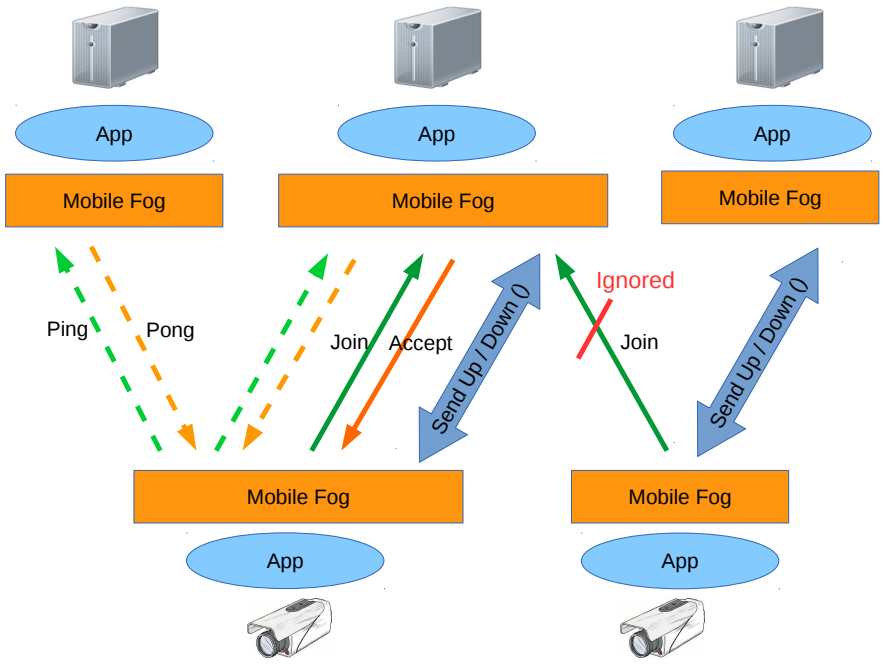


Figure 32: Two-phase Resource Discovery Protocol of Mobile Fog

Fog also takes communication latencies into account when selecting a parent node. Current implementation of Mobile Fog simply ignores "pong" messages if the round trip time between "ping" and corresponding "pong" messages is larger than the QoS requirement between two application components.

While simple and scalable, such a resource discovery protocol with a single phase of message exchange (i.e., "ping" and "pong" messages) may cause high end-to-end latencies due to sudden workload increases. For example, many child nodes may select the same parent node at the same time, since the parent node is currently not overloaded. If too many child nodes start streaming to the same parent, workload level at the parent node will suddenly increase, causing high latencies for stream processing until workload-driven adaptation happens. To avoid this problem, Mobile Fog uses a two-phase resource discovery protocol with explicit "join" and "accept" messages (Figure 32). With the two-phase protocol, a parent node can still acknowledge its availability with "pong" messages at any time, while selectively send "accept"

messages to prevent sudden increase of its workload level.

6.3.2 Incremental Application Deployment

To launch an application, a developer invokes *start_app()* with five parameters. The first parameter, *appkey*, specifies the application code to deploy at each level of the network hierarchy. The *region* parameter specifies a geospatial region where the application will run. The *level* parameter specifies the total number of levels in the application's logical hierarchy. For instance, a video surveillance application may have three levels: motion detection, face recognition, and spatio-temporal analysis. The *capacity* parameter specifies the class of computing instances needed by the application at each level of the network hierarchy. The last parameter, *QoS*, indicates communication latency requirements at each level of network hierarchy. Mobile Fog uses this parameter to find an appropriate upper-tier computing resources for hosting upper-level application components.

Using the dynamic resource discovery protocol, Mobile Fog incrementally maps the logical hierarchy of an application onto the physical hierarchy of the network infrastructure. Initially, when an application is launched, the application code (bundled in with the Mobile Fog runtime) runs on each stream source including mobile devices and sensors. When a new event is detected (e.g., a new target), the application calls send up() for further processing (e.g., target tracking), which triggers application deployment on an upper-tier computing resource. Similarly, the upper-tier computing resource may invoke send up() for the next processing step (e.g., face detection), resulting in application deployment at another upper-tier computing resource. Such incremental application deployment from the edge to the core resources allows highly adaptive resource utilization driven by application dynamics and QoS needs from the edge of the network.

Table 3: Mobile Fog API

Interface	Description
void send_up (message m)	Sends a message asynchronously from a child node to its parent node.
void send_down (message m)	Sends a message asynchronously from a parent node to its child nodes.
set<object>get_object (key k, location l, time t)	Get application data that matches a key, location (range), and time (range).
void put_object(object o, key k, location l, time t)	Put an application data associated with a key, location (range), and time (range).

6.4 API and Handlers

Mobile Fog helps an application developer to distribute workloads over physical network hierarchy (i.e., vertical workload distribution) through its hierarchical communication API as specified in Table 3. An application component can call *send_up()* when it wants to send a message to an upper-tier computing resource for further processing. If *send_up()* is called for the first time, Mobile Fog finds the right upper-tier computing resource and deploys the application using its dynamic resource discovery protocol. Once an application is deployed, subsequent invocations of *send_up()* will send asynchronous messages to the same parent node to support stateful stream processing. When a message arrives at the parent node, Mobile Fog invokes *on_send_up()* event handler to let application handles the message. Similarly, an application component running at the parent node can send a message to its child by invoking *send_down()*, and Mobile Fog will invoke *on_send_down()* at the child node when the message arrives.

Since many situation awareness applications perform stateful computation on continuous streams, Mobile Fog supports transferring an application state when it migrates a child node from its parent node to another parent node. Before starting the actual migration of a child node, Mobile Fog invokes the *on_migration_start ()*

Table 4: Mobile Fog Handlers

Handler	Description
void on_send_up (message m)	Called when a new message is arrived from a child node.
void on_send_down (message m)	Called when a new message is arrived from a parent node.
state on_migration_start ()	Called before a migration process starts. Application code running at the original parent node provides a stream context by returning a state object.
void on_migration_end (state s)	Called after a migration process ends. Application code running at a new parent node can recover a stream context from the provided state object.

handler to allow an application component to pack an application state for the child node into a single object. Mobile Fog then automatically transfers this object to the new parent node during the migration procedure. After the migration is complete (i.e., after resource discovery and application deployment), Mobile Fog invokes the *on_migration_end ()* handler at the new parent node to allow an application to initialize its state for the child node using the transferred object. Although these handlers support stateful stream processing with Mobile Fog's resource adaptation, Mobile Fog itself does not guarantee reliable stream processing when low-level system crashes (e.g., power off, virtual machine crashes, and so on) happen. We envision that the fog computing infrastructure will provide adequate system supports for such low-level reliability with virtualization technologies [3, 41, 45].

Mobile Fog also allows applications to share information across different streams through its spatio-temporal event API (Table 3). For example, an autonomous vehicle can generate live traffic information from its sensor streams (e.g., road constructions and car accidents) and store the information as a spatio-temporal event using *put_object*() interface. To access the event, other cars in proximity can all

get_object() interface that retrieves events matching spatio-temporal ranges from nearby computing resources. Since the API is used to store / retrieve live events, it is critical to support accessing those events with low latency. We discuss our future research direction to support this API in Section 7.3.2.

6.5 Latency- and Workload-driven Adaptation

Situation awareness applications need to deal with real world dynamics such as mobile stream sources and dynamic workloads from different areas. To meet QoS requirements of applications with such dynamics, Mobile Fog supports both workload- and latency-driven resource adaptation. For latency-driven adaptation, Mobile Fog monitors the round-trip latency between a child and a parent node. When the network latency becomes larger than a threshold, a child node rediscovers a new parent node using its resource discovery protocol as previously described. The child node keeps sending stream data to the original parent node until a new parent node is ready to take over.

For workload-driven adaptation, Mobile Fog monitors the workload at each node. Mobile Fog triggers a migration procedure when a certain number of deadline misses are recently detected. During the migration procedure, Mobile Fog sends migration requests to selected child nodes until the workload becomes lower than a threshold. During the migration procedure, the parent chooses the farthest child node (measured in round-trip latency) since this child is a likely candidate to move another parent due to application mobility. When a child node receives a migration request from its parent, it stops sending stream data to the parent node (*send_up()* messages are buffered in the Mobile Fog library) and starts the discovery procedure to find a new parent.

6.6 Use Case Analysis

This section qualitatively proves the efficacy of Mobile Fog using two application scenarios. In general, Mobile Fog automatically ensures quality of service requirements while handling real-world dynamics due to mobility and dynamic workloads.

6.6.1 Situation Awareness using a Distributed Camera Network

Mobile Fog can help to reduce latency while lightening the load on the core network by using resources located closer to the sensors. For example, imagine an intelligent surveillance application based on large-scale camera networks. The application runs on the fog computing infrastructure with three levels of network hierarchy including smart cameras, middle boxes attached to routers, and virtual machines in the cloud. Each smart camera monitoring a certain physical area performs motion detection algorithm, and only streams video frames that include motions using *send_up()* call. Using the motion frames, the application running at each middle box performs real-time target tracking for the moving objects. While tracking, the application can send down command messages to PTZ smart cameras to control their angles for better tracking on targets. The application also invokes *send_up()* call from middle boxes if face images are detected while tracking targets. Using the face images, the application running at the cloud performs a face recognition algorithm to determine the identities of occupants. The application at the cloud also sends down command messages to middle boxes to notify the priority of the target. Using this priority, the application running at the middle box can ensure real-time tracking on important targets.

Mobile Fog provides number of benefits in this application scenario. First of all, Mobile Fog allows the application to use nearby computing resources at middle boxes, achieving low latency and high bandwidth for streaming motion video frames for further processing. Secondly, Mobile Fog ensures low-latency for command messages, helping application logic to quickly react to the physical environment. Lastly, Mobile

Fog lowers bandwidth utilization at the core network since the large volume of motion frames are processed at the middle boxes and only relatively small-sized face images are sent up to the cloud.

6.6.2 Live Analysis for Autonomous Vehicles

With technological advances in sensors and analytics, autonomous vehicles become one of the most promising applications. While each vehicle performs real-time processing of local sensor data, it may also generate a personalized query[1] about the surrounding environment to figure out road conditions and the best route to a destination. Although each vehicle has local resources for processing local sensor data, such a query for surrounding environment may involve highly dynamic workload depending on the amount of traffic at a certain area. To deal with such dynamic workload, an autonomous vehicle can generate a query by invoking *send_up()* with query parameters. Based on the query parameters, the application running at a nearby computing node aggregates sensor data, perform analysis on the sensor data, then returns the result of live analysis to the vehicle using *send_down()* call.

In this application scenario, Mobile Fog helps the application to achieve low end-to-end latency for query processing. While vehicles are moving, Mobile Fog finds nearby computing resources with low network latency that can quickly provide query results. In addition to the latency- driven adaptation, Mobile Fog performs workload-driven adaptation to select the right parent nodes while the number of vehicles at a certain location changes over time.

6.7 Evaluation

In this section, we perform a set of experiments to show performance benefits of using Mobile Fog for situation awareness applications. In particular, we investigate

[1]Koldehofe, et al. [44] show the importance of consumer-specific operator graphs in autonomous vehicle applications.

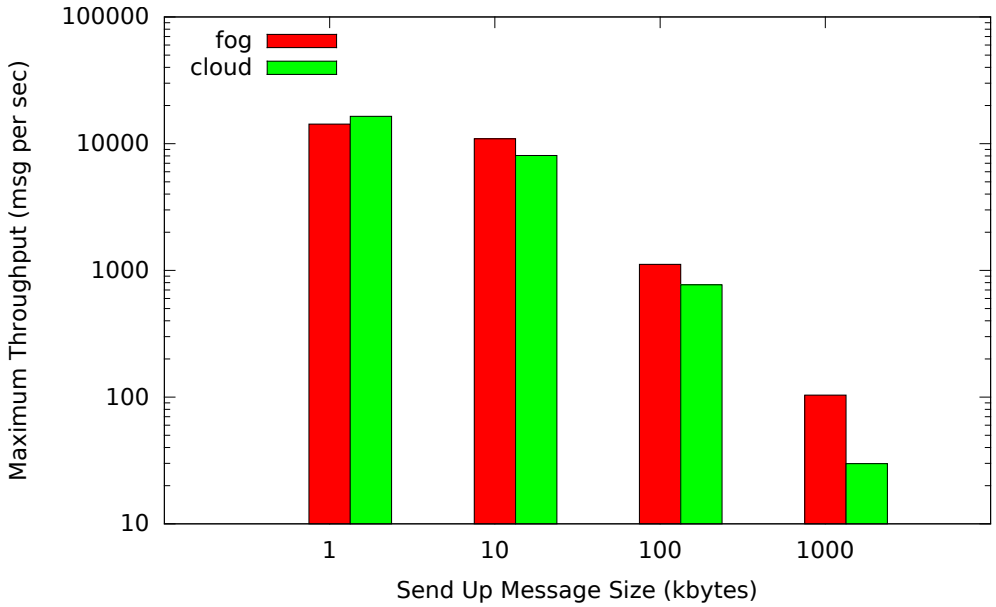


Figure 33: Throughput between a Client and an Upper-level Computing Resource

the throughput and end-to-end latency of situation awareness applications based on Mobile Fog. We also study the impact of Mobile Fog's resource adaptation mechanism on the application-level quality of service using realistic workloads.

6.7.1 Impact of Vertical Workload Distribution with Fog and Cloud

Mobile Fog supports vertical workload distribution across physical network hierarchy, allowing low-latency communication between distributed components of an application. To show the benefit of low-latency communication, we measured end-to-end latency and throughput of an application based on two different configurations: namely a fog setup and a cloud setup. The fog setup consists of a client and an upper-level computing resource that are in the same campus network[2], while the cloud setup consists of a client in the campus network and an upper-level computing resource in Amazon EC2 at Virginia. In the application code, the client reports a dummy event by invoking *send_up()* with different sizes of payloads. The client calls *send_up()* with

[2]Both the client and the upper-level computing resource are connected to Georgia Tech's campus network through Ethernet.

best effort (i.e., keeps calling *send_up()* with no delay) for throughput measurement but calls *send_up()* every second for latency measurement. The upper-level computing resource simply responds to the dummy event by calling *send_down()* with a one kilobytes of payload in *on_send_up()* handler. The average pure network latency measured using *Ping* between the client and the upper-level computing resource is 0.89 milliseconds in the fog setup and 15.92 milliseconds in the cloud setup[3].

Figure 33 shows the throughput of an application in terms of the number of responds arrived at the client per second. As shown in the figure, both cloud and fog setup yield similar throughput when payloads are small (up to 100 kilobytes). However, when the payload size is large (one megabyte for each *send_up()*), the throughput difference between the fog and the cloud setup becomes significant. This is because the communication latency between two application components decide the TCP throughput between those components. As the result suggest, it is critical to provide low communication latency for distributed components if they exchange messages with significant sizes of payloads.

Figure 34 shows the round-trip latency between the *send_up()* call and the corresponding *on_send_down()* handler in the client. In the figure, each bar represents median latency of each setup, while a low error bar indicates 25th percentile latency and a high error bar indicates 95th percentile latency. As the figure shows, the end-to-end latency is always much lower in the fog setup, allowing real-time interaction between distributed components in an application. The figure also shows that the fog setup provides less jitters in end-to-end latencies between those components. For a large payload size (one megabytes), however, both configurations show similar jitterness since end-to-end latency is driven by throughput for larger messages.

[3]Georgia Tech's campus network and Amazon EC2 are connected through Internet 2, providing the best case scenario. The latency can be much higher for other access networks such as 3G/4G mobile networks.

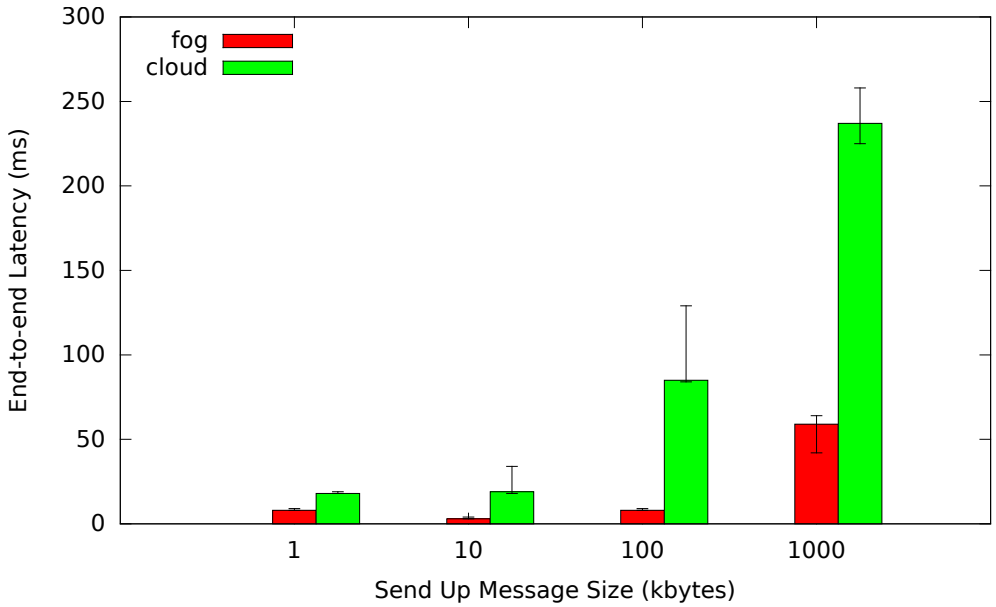


Figure 34: End-to-end Round Trip Latency between a Client and an Upper-level Computing Resource

6.7.2 Impact of Horizontal Workload Distribution with Fog Resources

Mobile Fog performs workload-driven resource adaptation to horizontally distribute application workload across nearby computing resources at the same level of network hierarchy. To show the impact of such horizontal workload distribution on application performance, we set up four heterogeneous computing resources that are placed in the same campus network with clients that are running on a single physical resource[4]. In the application code, a client reads a video file and calls *send_up()* with a video frame for every second. An upper-level computing resource detects faces using Viola-Jones face detection algorithm [70] from the OpenCV [9] library. After detecting faces from a video frame, the upper-level computing resource returns a message to a client by calling *send_down()* with the number of faces detected from the frame.

Based on the setup, we measured the throughput and deadline misses of the application using Mobile Fog with two different resource discovery protocols: single-phase

[4]We make sure that the single physical resource is not overloaded while running multiple clients.

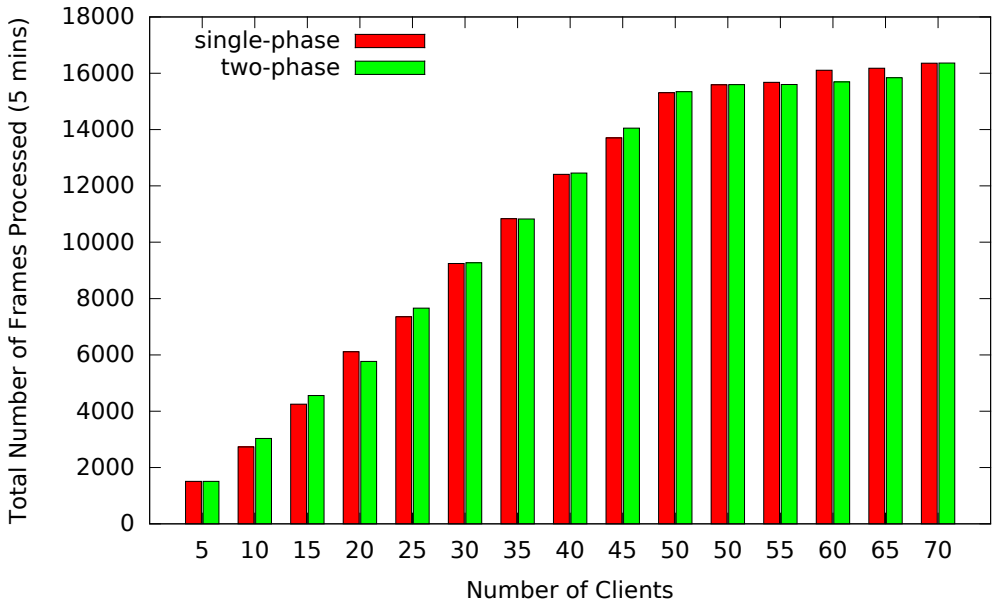


Figure 35: Throughput of Face Detection using Mobile Fog

discovery and two-phase discovery protocols. The single-phase discovery protocol only uses "ping" and "pong" messages between child and parent nodes while two-phase discovery protocol additionally uses "join" and "accept" messages to avoid sudden increases of workloads. Figure 35 presents the throughput of the application while increasing the number of clients. As shown in the figure, throughput increases linearly until all computing resources are overloaded with 50 clients. The experimental result indicates that Mobile Fog's workload-driven adaptation with both resource discovery protocols can fully utilize nearby computing resources without additional overheads.

Since many situation awareness applications need to provide actionable knowledge in timely manner, it is critical to process live stream data within a certain latency bound. Figure 36 shows the number of deadline misses for detecting faces from video frames using Mobile Fog. We set the deadline as two seconds based on the the average processing latency (about 500 milliseconds) for face detection on computing resources. This means that a new frame arrived at a single-core resource will result in a deadline miss if more than three frames are waiting in a queue. As shown in the figure, both

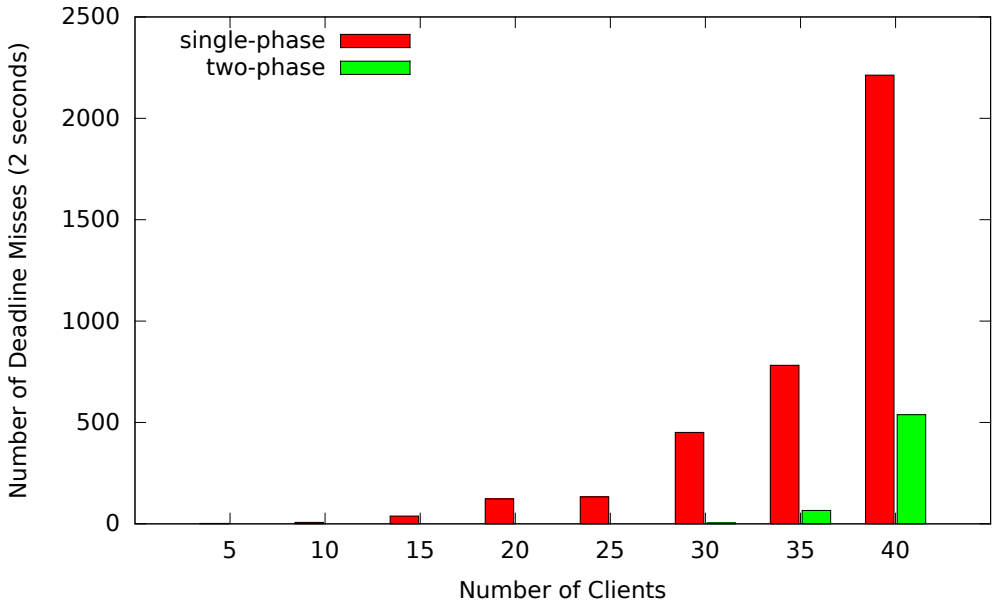


Figure 36: Number of Deadline Misses of Face Detection using Mobile Fog

protocols cause more deadline misses for more clients because Mobile Fog's resource adaptation happens more frequently when workload levels are higher on computing resources[5]. Compared to the two-phase protocol, the single-phase discovery protocol causes much more deadline misses because it involves sudden increases of workloads when multiple child nodes simultaneously select the same parent node. The experimental result suggests that Mobile Fog should use two-phase resource discovery protocol to ensure QoS requirements while dealing with dynamic workloads.

6.8 Conclusion

The rise of Internet of Things (IoT) enables various situation awareness applications on sensors and mobile devices (e.g., a smart surveillance system and autonomous vehicles). Due to the inherent dynamism of data streams from IoT devices, situation awareness applications need to handle dynamic workloads over space and time. Although existing datacenter-based clouds provide elastic computing resources, they

[5]Although the number of clients does not change during an experiment, various processing latencies for processing different frames result in workload-driven resource adaptation.

cannot support latency-sensitive quality of service since data streams need to go through the Internet to reach computing resources. Cisco recently proposed a new computing paradigm, fog computing, that provides geospatially distributed computing resources at different levels of network hierarchy. While the fog computing infrastructure potentially supports large-scale, latency-sensitive applications on highly dynamic IoT devices, managing distributed resources becomes a new challenge due to the complexity of fog computing resources.

To solve the problem, we developed a fog-based execution environment called Mobile Fog. To provide low-latency communication between distributed components of an application, Mobile Fog automatically deploys those components onto computing resources at different levels of network hierarchy, and allow them to communicate each other using hierarchical communication API. Mobile Fog also performs automatic resource adaptation to ensure latency-sensitive quality of service while dealing with dynamic workloads and mobility. Through a set of experiments, we evaluated the impact of vertical and horizontal workload distribution using Mobile Fog and proved that Mobile Fog ensures latency-sensitive quality of service with highly dynamic workloads.

CHAPTER VII

DISCUSSION AND FUTURE DIRECTION

This chapter discusses several insights from our work and presents possible directions for the future work. In particular, we discuss the workload characteristic of situation awareness applications and various approaches we used to reduce end-to-end latencies for situation awareness. For the future work, we present possible research directions in each part of our distributed framework, including programming models and runtime mechanisms.

7.1 Dynamic Workloads of Situation Awareness Applications

Situation awareness applications on camera networks have highly dynamic workloads that depend on the real-world situations. Specifically, video analytics used by those applications have dynamic workloads for processing each video stream, depending on the content of a video stream. Unlike other modalities of sensors that generate structured data streams (e.g., RFID, temperature, light, motion sensors), a video stream may contain multiple independent events and objects. As the number of events and objects in a video stream changes frequently depending on real-world situations, processing each video stream involves highly dynamic workload.

In addition to the dynamic workload of each individual stream, the number of camera streams also frequently changes over space and time in applications using mobile devices. For instance, autonomous vehicle applications have dynamic workloads depending on the number of vehicles in an area. To support such dynamic workloads of applications, Mobile Fog provides decentralized resource discovery and adaptation mechanisms that automatically migrates streams based on real-time workloads.

7.2 Approaches to Reduce End-to-end Latency for Situation Awareness

Since situation awareness applications have latency-sensitive quality of service, it is critical to provide actionable knowledge with low latency. Our work includes four different approaches that reduce end-to-end latencies for those applications. The first approach is to exploit fine-grained parallelism in a video stream. For instance, target tracking involves following targets in a video frame using previous positions and features of targets. To reduce the latency of processing a video frame, we can process those multiple targets simultaneously using parallel computing resources (e.g., GPU or multicore CPU).

Another approach is to predict a user query and start processing events before the query. This approach is useful in mobile situation awareness applications since mobile users make customized queries based on their geospatial locations. If we can generate highly accurate results before user arrival, we can provide just-in-time situation information without processing latency.

The next approach is to use computing resources in proximity for live stream analysis. Nearby computing resources provide low communication latency between stream sources and computing resources, allowing high throughput and low end-to-end latency for stream processing. Furthermore, the low communication latency is critical for controlling actuators such as robots to deal with real-world situations.

Lastly, we can trade off application-level fidelity for end-to-end latency. In spatio-temporal analysis, for example, we have proposed a selective state update mechanism to reduce communication cost among distributed workers. Although the approximate state is different from the original application state , we showed that query results based on the approximate state are still highly accurate for various types of queries. Since events from camera networks are inherently uncertain, such an approach can be used in various situation awareness applications processing camera network events.

7.3 Future Direction

Each part of this dissertation (i.e., programming models and runtime mechanisms) has a future research direction to further improve performance or to support broader scope of applications.

7.3.1 Dynamic Workload Distribution across Network Hierarchy

Mobile Fog dynamically distributes application workloads over nearby computing resources (dynamic horizontal distribution), but it does not migrate workloads over different levels of network hierarchy (static vertical distribution). In some cases, however, dynamic workload distribution across network hierarchy is critical to meet the quality of service. For example, the network latency from a certain level to the upper level of network hierarchy varies depending on the physical location of computing resources. To support mobile devices with such dynamic latencies between different levels of network hierarchy, Mobile Fog needs to find the right computing resource across both geospatial space and network hierarchy. More importantly, all nearby computing resources may be fully overloaded due to the excessive workloads from the real-world. Since geospatially faraway resources at the same network hierarchy would involve high latencies, Mobile Fog needs to dynamically migrate some workloads to upper-tier computing resources.

7.3.2 Efficient Spatio-temporal Event Storage using Fog

Situation awareness applications often require accessing events that are associated with space and time properties. For instance, different autonomous vehicles in the same area may want to share information of the latest road condition (e.g., road constructions and car accidents). Since situation awareness applications have latency-sensitive quality of service, it is critical to provide such information with low latency. In our opportunistic event processing mechanism, we proposed a way to reduce computational latency for generating situational information. However, it is also critical

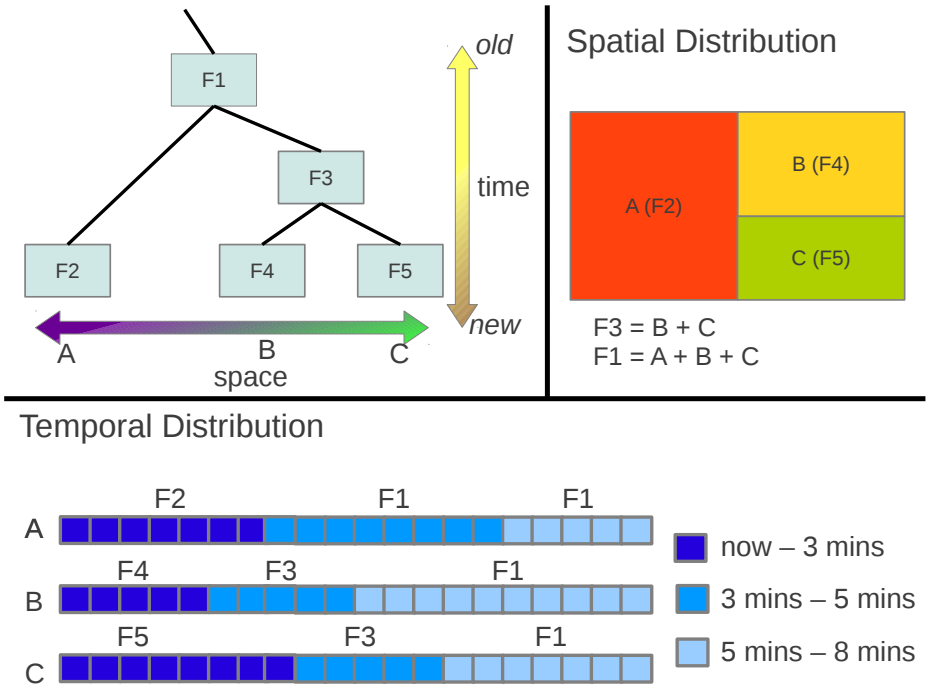


Figure 37: Distributed Spatio-temporal Event Storage using Fog

to efficiently store / retrieve situational information with low latency.

One possible approach to achieve low-latency event storing and retrieval is to place interested events near users. Specifically, we can build an efficient spatio-temporal event storage by exploiting the fact that mobile users are mostly interested in live events about their surrounding areas. As Figure 37 shows, we can place event data based on their temporal and spatial properties. Since recent events will be accessed more frequently by mobile users, we place the live events near the edge of the network infrastructure while placing older events at the core of the network infrastructure. We also place events at geospatially nearby computing resources based on their spatial properties, allowing mobile users to quickly access events about their surrounding areas.

An interesting challenge in this direction is to support consistent latencies for retrieving events while dealing with dynamic event generation rates at different locations. If mobile users are interested in recent five minutes of events, it would be ideal

to store the five minutes of events at every edge computing resource. However, the capacity of each computing resource is limited while event generation rate is highly dynamic, making it hard to maintain the consistent temporal range of live events at every computing resource. To solve the problem, the system must actively migrate events across distributed computing resources to balance temporal ranges of events.

7.3.3 Probabilistic Equality Comparison in Target Container

Current Target Container supports binary equality checker that returns either true or false for target equality. However, many feature comparison algorithms in computer vision domain are probabilistic, providing similarity between two features rather than exact true or false. To support those probabilistic algorithms, Target Container must provide updated API and priority-aware resource management. In particular, current priority-aware resource management assigns the same priority to different targets if they belong to a single physical target. If we use probabilistic equality checker, however, we must calculate the priority of individual targets in different streams based on similarities between different targets and their own local priorities. Note that such probabilistic equality comparison does not exclude those applications using binary equality comparison, since application developers can use only zero or one probability to implement binary equality comparison.

7.3.4 Event Ordering for Spatio-temporal Analysis

Spatio-temporal analysis requires events that are ordered based on their physical timestamps to maintain accurate application state. For example, suppose occupant A entered to zone X followed by occupant B entering zone Y. To preserve application-level correctness, it is necessary to perform the state update for event A before event B at every distributed state worker. However, there is no way to guarantee such ordering, given the vagaries of the Internet and the fact that our middleware has no control on the scheduling of the virtual computing resources allocated at a computing

infrastructure.

One possible solution to solve the problem is to buffer events for a certain amount of time before using them for state update. This approach allows reordering events within the time bound, but it increases end-to-end latency of situation awareness applications since it delays processing events. If inter-arrival time between spatio-temporal queries is larger than the time bound (i.e., end-to-end latency requirement is larger than the buffering time), however, we can provide accurate application state without hurting QoS requirement.

Another approach is to go back to an old application state and redo state update from the old state when an out-order-event is detected. Although this approach may not affect to the end-to-end latency of applications in normal operation, the cost of state recovery may be expensive depending on the frequency of our-order-events and the size of application state.

CHAPTER VIII

CONCLUSION

Technological advances in sensors and various analytics have enabled a new class of applications, situation awareness applications, that automatically generate actionable knowledge by processing live streams from widely deployed sensors. Despite of potential benefits, development complexity due to the dynamic nature of those applications and latency-sensitive quality of service has prevented those applications being used in various domains, including surveillance, entertainment, traffic monitoring, health care, and so on.

This dissertation proposes a distributed framework that enables those innovative applications on large numbers of sensing devices and computing resources. In particular, the framework provides two programming abstractions for different levels of application logic: multi-camera target tracking and spatio-temporal analysis. With the programming abstractions, application developers can simply provide their application logic in domain-specific handlers, while the backing runtime system automatically ensures performance issues of applications using parallel / distributed computing resources.

This work also presents two runtime mechanisms for low end-to-end latency and dynamic workload handling of situation awareness applications. The first mechanism, opportunistic event processing, processes events in predicted query regions to provide highly accurate just-in-time situational information to mobile users. The second mechanism, Mobile Fog, serves a fog-based execution environment that runs a large-scale situation awareness application on widely distributed computing resources, while performing both latency- and workload-driven adaptation.

```
void Detector(CAM cam, IMAGE img)
{
  DetectorData dd = TC_read_detector_data(cam);
  list<Tracker> tracker_list = dd.tracker_list;
  Truckled td;
  bool is_new;

  List<Blob> new_blob_list;
  List<Blob> old_blob_list;

  for_each(tracker in tracker_list)
  {
    td = TC_read_tracker_data(tracker);
    old_blob_list.add( td.blob );
  }

  new_blob_list = detect_blobs(img);

  for_each(nB in new_blob_list)
  {
    is_new = true;

    for_each(oB in old_blob_list
      If( blob_overlap(nB, oB) == TRUE)
        is_new = false;

    if(is_new == TRUE)
    {
      TrackerData new_td;
      new_td.blob = nB;
      TCData new_tcd;
      tcd.hist = calc_hist(img, nB);
      Tracker tracker = TC_create_target(tcd, td);
      dd.tracker_list.insert(tracker);
    }
  }
}
```

Figure 38: Example Detector

```
void Tracker(Tracker tracker, TC tc, CAM cam, IMAGE img)
{
  TrackerData td = TC_read_tracker_data(tracker);
  TCData tcd = TC_read_tc_data(tc);
  Histogram hist;
  int threat_level;
  Blob new_blob;

  new_blob = color_track(img, td.blob);

  If( is_out_of_FOV( new_blob ) )
    TC_stop_track(tracker, cam);

  data.blob = new_blob;
  TC_update_tracker_data(tracker, data);

  threat_level = calc_threat_level(img, new_blob);
  TC_set_priority(tc, threat_level);

  hist = calc_hist(img, new_blob);

  if( compare_hist( hist, tcd.hist) < CHANGE_THRES )
  {
    tcd.hist = hist;
    TC_update_tc_data(tc, tcd);
  }
}
```

Figure 39: Example Tracker

```
bool Equality_checker(TCData src1, TCData src2)
{
  if( compare_hist( src1.hist, src2.hist) > EQUAL_THRES )
    return TRUE;

  return FALSE;
}

void Merger(TCData src1, TCData src2, TCData dst)
{
  dst.hist = average_hist(tcd1.hist, tcd2.hist);
}
```

Figure 40: Example Equality Checker and Merger

REFERENCES

[1] Adi, A. and Etzion, O., "Amit - the situation manager," *The VLDB Journal*, vol. 13, pp. 177–203, May 2004.

[2] Akidau, T., Balikov, A., Bekiroğlu, K., Chernyak, S., Haberman, J., Lax, R., McVeety, S., Mills, D., Nordstrom, P., and Whittle, S., "Millwheel: fault-tolerant stream processing at internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A., "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[4] Behrisch, M., Bieker, L., Erdmann, J., and Krajzewicz, D., "Sumo-simulation of urban mobility-an overview," in *SIMUL 2011, The Third International Conference on Advances in System Simulation*, pp. 55–60, 2011.

[5] Bobick, A. F. and Davis, J. W., "The recognition of human movement using temporal templates," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 23, no. 3, pp. 257–267, 2001.

[6] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S., "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, MCC '12, (New York, NY, USA), pp. 13–16, ACM, 2012.

[7] Borcea, C., Intanagonwiwat, C., Kang, P., Kremer, U., and Iftode, L., "Spatial programming using smart messages: Design and implementation," in *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pp. 690–699, IEEE, 2004.

[8] Bouchaffra, D. and Govindaraju, V., "A methodology for mapping scores to probabilities," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 21, pp. 923 –927, sep 1999.

[9] Bradski, G. and Kaehler, A., *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008.

[10] Brumitt, B., Meyers, B., Krumm, J., Kern, A., and Shafer, S., "Easyliving: Technologies for intelligent environments," in *Handheld and ubiquitous computing*, pp. 12–29, Springer, 2000.

[11] Chen, P., Ahammad, P., Boyer, C., Huang, S.-I., Lin, L., Lobaton, E., Meingast, M., Oh, S., Wang, S., Yan, P., Yang, A., Yeo, C., Chang, L.-C., Tygar, J., and Sastry, S., "Citric: A low-bandwidth wireless camera network platform," in *Distributed Smart Cameras, 2008. ICDSC 2008. Second ACM/IEEE International Conference on*, pp. 1 –10, sept. 2008.

[12] Clinch, S., Harkes, J., Friday, A., Davies, N., and Satyanarayanan, M., "How close is close enough? understanding the role of cloudlets in supporting display appropriation by mobile users," in *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on*, pp. 122–127, IEEE, 2012.

[13] Comaniciu, D., Ramesh, V., and Meer, P., "Real-time tracking of non-rigid objects using mean shift," *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, vol. 2, p. 2142, 2000.

[14] Constine, J., "Shopper tracker's kinect hack is like google analytics for retail store shelves," December 2011. [Online; posted 5-December-2011; `http://techcrunch.com/2011/12/05/shopper-tracker-kinect/`].

[15] Cugola, G. and Margara, A., "Tesla: a formally defined event specification language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, (New York, NY, USA), pp. 50–61, ACM, 2010.

[16] Cugola, G. and Margara, A., "Low latency complex event processing on parallel hardware," *Journal of Parallel and Distributed Computing*, vol. 72, no. 2, pp. 205–218, 2012.

[17] Dilley, J., Maggs, B., Parikh, J., Prokop, H., Sitaraman, R., and Weihl, B., "Globally distributed content delivery," *Internet Computing, IEEE*, vol. 6, no. 5, pp. 50–58, 2002.

[18] du Mouza, C., Litwin, W., and Rigaux, P., "SD-Rtree: A Scalable Distributed Rtree," in *Proc. of IEEE 23rd International Conference on Data Engineering*, ICDE 2007, pp. 296–305, Apr. 2007.

[19] Elgammal, A., Duraiswami, R., Harwood, D., and Davis, L. S., "Background and foreground modeling using non-parametric kernel density estimation for visual surveillance," *Proceedings of the IEEE*, July 2002.

[20] Elgammal, A., Harwood, D., and Davis, L., "Non-parametric model for background subtraction," in *Computer Vision-ECCV 2000*, pp. 751–767, Springer, 2000.

[21] Feris, R., Hampapur, A., Zhai, Y., Bobbitt, R., Brown, L., Vaquero, D., Tian, Y., Liu, H., and Sun, M.-T., "Case-Study: IBM smart surveillance system," in *Intelligent Video Surveillance: Systems and Technologies* (Ma, Y. and Qian, G., eds.), Taylor & Francis, CRC Press, 2009.

[22] Gavrila, D. M., "The visual analysis of human movement: A survey," *Computer vision and image understanding*, vol. 73, no. 1, pp. 82–98, 1999.

[23] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M., "SPADE: the System S declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, (New York, NY, USA), pp. 1123–1134, ACM, 2008.

[24] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M., "Spade: the system s declarative stream processing engine," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1123–1134, ACM, 2008.

[25] Gropp, W., Lusk, E., and Skjellum, A., *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.

[26] Güting, R. H., Böhlen, M. H., Erwig, M., Jensen, C. S., Lorentzos, N. A., Schneider, M., and Vazirgiannis, M., "A foundation for representing and querying moving objects," *ACM Transactions on Database Systems (TODS)*, vol. 25, no. 1, pp. 1–42, 2000.

[27] Haklay, M. and Weber, P., "Openstreetmap: User-generated street maps," *Pervasive Computing, IEEE*, vol. 7, pp. 12 –18, oct.-dec. 2008.

[28] Hampapur, A., Brown, L., Connell, J., Ekin, A., Haas, N., Lu, M., Merkl, H., and Pankanti, S., "Smart video surveillance: exploring the concept of multiscale spatiotemporal tracking," *Signal Processing Magazine, IEEE*, vol. 22, pp. 38 – 51, march 2005.

[29] Haritaoglu, I., Harwood, D., and Davis, L., "W4: Who? when? where? what? a real time system for detecting and tracking people," *Automatic Face and Gesture Recognition, IEEE International Conference on*, vol. 0, p. 222, 1998.

[30] Hendawi, A. M. and Mokbel, M. F., "Panda: A Predictive Spatio-Temporal Query Processor," in *Proceedings of the 20th International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '12, (New York, NY, USA), pp. 13–22, ACM, 2012.

[31] Hirzel, M., "Partition and compose: parallel complex event processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, (New York, NY, USA), pp. 191–200, ACM, 2012.

[32] Hong, K., Lillethun, D., Ramachandran, U., Ottenwälder, B., and Koldehofe, B., "Mobile fog: a programming model for large-scale applications on the internet of things," in *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pp. 15–20, ACM, 2013.

[33] Hong, K., Lillethun, D., Ramachandran, U., Ottenwälder, B., and Koldehofe, B., "Opportunistic spatio-temporal event processing for mobile situation awareness," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pp. 195–206, ACM, 2013.

[34] Hong, K., Ottenwälder, B., and Ramachandran, U., "Scalable spatio-temporal analysis on distributed camera networks," in *Intelligent Distributed Computing VII*, pp. 131–140, Springer, 2014.

[35] Hong, K., Smaldone, S., Shin, J., Lillethun, D. J., Iftode, L., and Ramachandran, U., "Target container: A target-centric parallel programming abstraction for video-based surveillance," in *ICDSC*, pp. 1–8, 2011.

[36] Hong, K., Voelz, M., Govindaraju, V., Jayaraman, B., and Ramachandran, U., "A distributed framework for spatio-temporal analysis on large-scale camera networks," Tech. Rep. GT-CS-12-10, Georgia Institute of Technology.

[37] Hope, C., "1,000 cctv cameras to solve just one crime, met police admits," August 2009. [Online; posted 25-August-2009; `http://www.telegraph.co.uk/news/uknews/crime/6082530/1000-CCTV-cameras-to-solve-just-one-crime-Met-Police-admits.html`].

[38] Jeung, H., Yiu, M. L., Zhou, X., and Jensen, C. S., "Path prediction and predictive range querying in road network databases," *The VLDB Journal*, vol. 19, pp. 585–602, Aug. 2010.

[39] Johnson, D. S., "Approximation algorithms for combinatorial problems," in *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, (New York, NY, USA), pp. 38–49, ACM, 1973.

[40] Kale, A., Cuntoor, N., Yegnanarayana, B., Rajagopalan, A., and Chellappa, R., "Gait analysis for human identification," in *Audio-and Video-Based Biometric Person Authentication*, pp. 1058–1058, Springer, 2003.

[41] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A., "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, vol. 1, pp. 225–230, 2007.

[42] Koch, G. G., Koldehofe, B., and Rothermel, K., "Cordies: expressive event correlation in distributed systems," in *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, (New York, NY, USA), pp. 26–37, ACM, 2010.

[43] Koldehofe, B., Dürr, F., Tariq, M. A., and Rothermel, K., "The power of software-defined networking: line-rate content-based routing using openflow," in *Proceedings of the 7th Workshop on Middleware for Next Generation Internet Computing*, p. 3, ACM, 2012.

[44] Koldehofe, B., Ottenwälder, B., Rothermel, K., and Ramachandran, U., "Moving Range Queries in Distributed Complex Event Processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, (New York, NY, USA), pp. 201–212, ACM, 2012.

[45] Koslovski, G., Yeow, W.-L., Westphal, C., Huu, T. T., Montagnat, J., and Vicat-Blanc, P., "Reliability support in virtual infrastructures," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pp. 49–58, IEEE, 2010.

[46] Kulkarni, P., Ganesan, D., Shenoy, P., and Lu, Q., "Senseye: a multi-tier camera sensor network," in *Proceedings of the 13th annual ACM international conference on Multimedia*, pp. 229–238, ACM, 2005.

[47] Li, G. and Jacobsen, H.-A., "Composite subscriptions in content-based publish/subscribe systems," in *Middleware '05: Proceedings of the ACM/IFIP/USENIX 2005 International Conference on Middleware*, (New York, NY, USA), pp. 249–269, Springer-Verlag New York, Inc., 2005.

[48] Lillethun, D. J., Hilley, D., Horrigan, S., and Ramachandran, U., "MB++: An integrated architecture for pervasive computing and high-performance computing," in *Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, 2007.

[49] Little, J. and Boyd, J. E., "Recognizing people by their gait: The shape of motion," *Videre*, vol. 1, pp. 1–32, 1996.

[50] Luo, L., Abdelzaher, T. F., He, T., and Stankovic, J. A., "Envirosuite: An environmentally immersive programming framework for sensor networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 5, no. 3, pp. 543–576, 2006.

[51] Menon, V., Jayaraman, B., and Govindaraju, V., "Multimodal identification and tracking in smart environments," *Personal Ubiquitous Comput.*, vol. 14, pp. 685–694, December 2010.

[52] Menon, V., Jayaraman, B., and Govindaraju, V., "The three rs of cyber-physical spaces," *Computer*, vol. 44, no. 9, pp. 73–79, 2011.

[53] Neumeyer, L., Robbins, B., Nair, A., and Kesari, A., "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pp. 170–177, IEEE, 2010.

[54] Nickolls, J., Buck, I., Garland, M., and Skadron, K., "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[55] Padhye, J., Firoiu, V., Towsley, D., and Kurose, J., "Modeling tcp throughput: A simple model and its empirical validation," in *ACM SIGCOMM Computer Communication Review*, vol. 28, pp. 303–314, ACM, 1998.

[56] Parag, T., Elgammal, A., and Mittal, A., "A framework for feature selection for background subtraction," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2006.

[57] Pfoser, D., Jensen, C. S., and Theodoridis, Y., "Novel Approaches in Query Processing for Moving Object Trajectories," in *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, (San Francisco, CA, USA), pp. 395–406, Morgan Kaufmann Publishers Inc., 2000.

[58] Pietzuch, P., Ledlie, J., Shneidman, J., Roussopoulos, M., Welsh, M., and Seltzer, M., "Network-aware operator placement for stream-processing systems," in *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, (Washington, DC, USA), pp. 49–, IEEE Computer Society, 2006.

[59] Pietzuch, P. R., Shand, B., and Bacon, J., "Composite event detection as a generic middleware extension," *Network, IEEE*, vol. 18, pp. 44 – 55, jan/feb 2004.

[60] Pillai, P. S., Mummert, L. B., Schlosser, S. W., Sukthankar, R., and Helfrich, C. J., "Slipstream: scalable low-latency interactive perception on streaming data," in *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video*, pp. 43–48, ACM, 2009.

[61] Qian, Z., He, Y., Su, C., Wu, Z., Zhu, H., Zhang, T., Zhou, L., Yu, Y., and Zhang, Z., "Timestream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 1–14, ACM, 2013.

[62] Reinders, J., *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[63] Rizou, S., Diirr, F., and Rothermel, K., "Fulfilling end-to-end latency constraints in large-scale streaming environments," *IEEE International Performance Computing and Communications Conference*, vol. 0, pp. 1–8, 2011.

[64] Satyanarayanan, M., Bahl, P., Caceres, R., and Davies, N., "The case for VM-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, October - December 2009.

[65] Schling, B., *The boost C++ libraries*. Xml Press, 2011.

[66] Shin, J., Kumar, R., Mohapatra, D., Ramachandran, U., and Ammar, M., "ASAP: A camera sensor network for situation awareness," in *OPODIS'07: Proceedings of 11th International Conference On Principles Of Distributed Systems*, 2007.

[67] Stauffer, C. and Grimson, W., "Adaptive background mixture models for real-time tracking," *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, vol. 2, p. 2246, 1999.

[68] Thies, W., Karczmarek, M., and Amarasinghe, S., "Streamit: A language for streaming applications," in *Compiler Construction*, pp. 179–196, Springer, 2002.

[69] Turk, M. and Pentland, A., "Eigenfaces for recognition," *Journal of cognitive neuroscience*, vol. 3, no. 1, pp. 71–86, 1991.

[70] Viola, P. and Jones, M. J., "Robust real-time face detection," *International journal of computer vision*, vol. 57, no. 2, pp. 137–154, 2004.

[71] Wang, L., Tan, T., Ning, H., and Hu, W., "Silhouette analysis-based gait recognition for human identification," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 25, no. 12, pp. 1505–1518, 2003.

[72] Welsh, M. and Mainland, G., "Programming sensor networks using abstract regions.," in *NSDI*, vol. 4, pp. 3–3, 2004.

[73] Whitehouse, K., Zhao, F., and Liu, J., "Semantic streams: A framework for composable semantic interpretation of sensor data," in *Wireless Sensor Networks*, pp. 5–20, Springer, 2006.

[74] Wilson, J., "Apps know the best hotspots for hookups," June 2011. [Online; posted 17-June-2011; `http://www.cnn.com/2011/TECH/mobile/06/17/bar.scene.apps/index.html`].

[75] Wold, S., Esbensen, K., and Geladi, P., "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1, pp. 37–52, 1987.

[76] Wolfson, O., Sistla, A. P., Chamberlain, S., and Yesha, Y., "Updating and querying databases that track mobile units," in *Mobile Data Management and Applications*, pp. 3–33, Springer, 1999.

[77] Wren, C., Azarbayejani, A., Darrell, T., and Pentland, A., "Pfinder: Real-time tracking of the human body," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, pp. 780–785, 1997.

[78] Zhao, W., Chellappa, R., Phillips, P. J., and Rosenfeld, A., "Face recognition: A literature survey," *Acm Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 399–458, 2003.